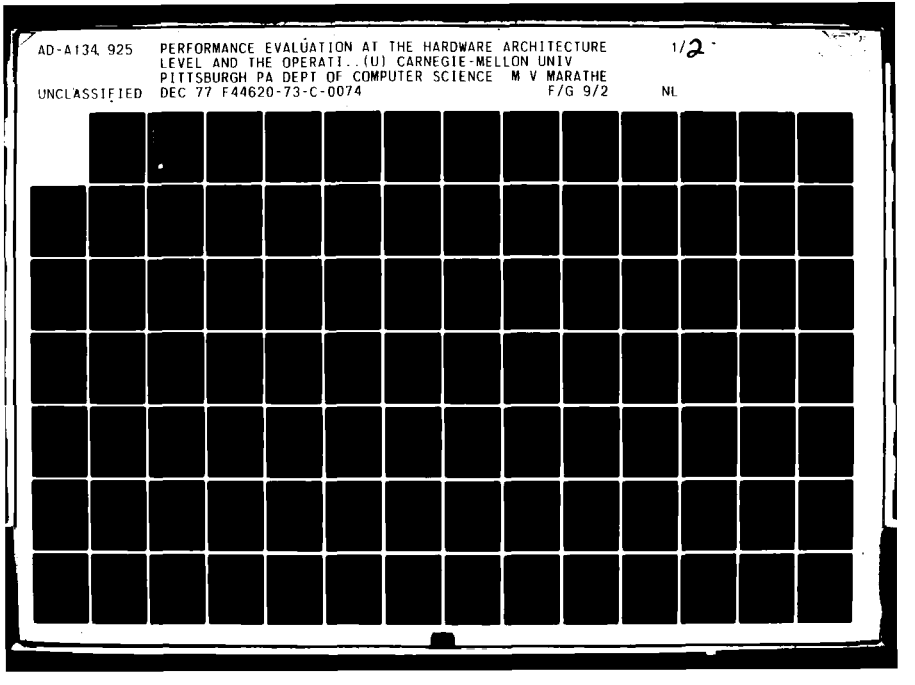
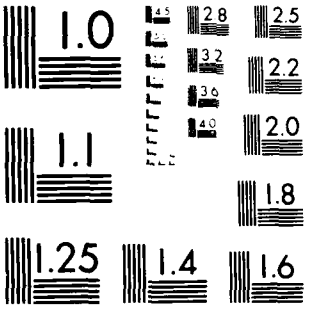

AD-A134 925
PERFORMANCE EVALUATION AT THE HARDWARE ARCHITECTURE LEVEL AND THE OPERATI..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE M V MARATHE
1/2
UNCLASSIFIED
DEC 77 F44620-73-C-0074
F/G 9/2
NL



PERFORMANCE EVALUATION AT THE HARDWARE ARCHITECTURE LEVEL AND THE OPERATING SYSTEM KERNEL DESIGN LEVEL

MADHAV V. MARATHE

DEPARTMENT of COMPUTER SCIENCE

Carnegie-Mellon University

PERFORMANCE EVALUATION AT THE HARDWARE ARCHITECTURE LEVEL AND THE OPERATING SYSTEM KERNEL DESIGN LEVEL

MADHAV V. MARATHE

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

December, 1977

This report reproduces a dissertation, titled as above, submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defence(contract F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

ABSTRACT

This thesis investigates the measurement and analysis of complex computer systems at the levels of hardware architecture and operating system kernel design. Both these levels provide the user with a set of instructions: the machine instructions and the operating system kernel functions. Our viewpoint is always that of the designers attempting to study the usage of these abstract instructions by measuring the behavior of actual programs. We take the same approach for the study of the effects of multiprocessing at these levels.

For the hardware architect, the variability in the workload has always been a difficult design problem. It seems intuitively clear that different application areas (scientific, business, process control) present different workloads to a processor. The important questions faced by the designer in this respect are: are the application areas different at the lowest level of data structure manipulations i.e. the instruction mix level? If so, are they sufficiently different to justify a specialized processor for each application area? How much performance improvement can be obtained by such specialized processors across all the programs in a given area? We apply statistical experimental design techniques to quantify the variance in the instruction mix due to the various factors in order to answer these questions. Our results indicate that the variance due to different programs within an area is comparable to the variance across application areas themselves. This shows that the differences across the application areas are not significant at the instruction mix level. This is a consequence of the fact that the machine instructions operate on bits and words whereas the operations on

higher level data structures such as vectors, process control blocks, queues and lists differentiate between the application areas.

In the case of multiprocessors, the study of the contention for shared resources among processors is very important. At the hardware architecture level, the contention occurs for the shared memory and shared data paths. This problem has been studied earlier by others ([BHAN73], [MCCR73], [BASK76]) using analytical models. We have attempted to measure the memory contention for C.mmp - the Carnegie-Mellon University's multi miniprocessor. Our study was hampered by the lack of high resolution measurement tools.

The measurement of the workload and its variation for the operating system kernel design level is complicated by the fact that each operating system kernel has its own set of primitive functions and comparisons across different operating systems is not possible with our current understanding of operating systems. We have therefore decided to defer the general study of this problem. However, one aspect of the problem that can be attacked is the problem of software lockout in a multiprocessor operating system.

In order to maintain integrity in a multiprocessor system, certain shared data objects (such as the list of runnable processes or the list of free blocks of memory) have to be accessed by only one processor at a time giving rise to software lockout. When two or more processors attempt to access the same shared data object at the same time, only one of them can access it and others have to wait. The mechanism used for such mutual exclusion is called a lock. The time lost by a processor while waiting for a shared object to become free can become a performance bottleneck for multiprocessors. A hardware monitor was used to measure the contention occuring in

Hydra, the operating system for C.mmp. The measurements show that while in the operating system, about 400 instructions are executed between successive locks and each locked execution takes about 100 instructions. However, the shared data of Hydra is organized into enough separate objects that very little time was lost due to contention for these objects. To the best of our knowledge an experimental investigation of this problem was not possible in the past. A simple central server queueing system was used to model the locking behavior and to predict the time lost due to contention. The predictions were validated against the actual measurements and the validated model was then used to predict time lost due to contention in larger systems. Our model predicts that time lost due to software lockout will not be significant for Hydra even when the number of processors in C.mmp is increased to 48. As other multiprocessor operating systems are designed, the model developed in our investigation can be used as a guide for the study of their software lockout problems.

The space of performance parameters at various system levels is examined and strengths of the various measurement tools are discussed with respect to the parameters. We also identify the hardware monitor as the primary measurement tool at the levels of hardware architecture and operating system kernel design. Because the hardware monitor is a versatile tool applicable to other levels as well, we have investigated many other applications of our hardware monitor, but to lesser depths. A survey of hardware monitoring techniques is also presented and suggestions are made regarding the design of future monitors.

A NOTE ON TERMINOLOGY

We have used the terminology introduced by Bell and Newell [BELL71] to describe computer structures throughout this dissertation. We list below the specific terms used along with their meaning:

ISP: The logical processor defined by its instruction set, as opposed to its physical implementation. The ISP of a processor includes such aspects as instruction formats, register structure, instruction interpretation algorithms, address calculation, data types and their representation.

PMS: This is the Processor, Memory and Switch notation developed by Bell and Newell for describing computer structures.

K.mon: This is the PMS name of our hardware monitor. It stands for a control element (K) for monitoring purposes hence K(monitor) or K.mon for short.

Host or P.host: This is the processor under measurement. The host computer has to be a PDP-11 in the current design of K.mon. K.mon is connected to the Unibus of the host processor.

Supervisor or P.sup: This is the processor controlling the K.mon. It sets up registers in K.mon, starts it and finally processes the data generated by K.mon. The supervisory processor is also constrained to be a PDP-11 in the current design of K.mon.

Hardware Architecture: This is the part of hardware design which deals with the ISP and its implementation.

Operating System Kernel: This is the level in the operating system where its most primitive operations are defined. A kernel includes the primitive synchronization mechanism, the protection mechanism, the low level resource allocation and scheduling functions.

Software lockout (also called software contention): This is a phenomenon arising out of the need for mutually exclusive accesses to shared data structures in a multiprocessor operating system. The software lockout results in a loss of of time while executing certain operating system functions.

ACKNOWLEDGEMENTS

First and foremost, I thank Professor Samuel Fuller for encouraging this research from its early stages and for contributing time and ideas all along. I am also grateful to him for reviewing innumerable versions of the draft. I am indebted to my thesis committee- Professors Jack McCredie, Alice Parker and William Wulf who helped shape the final form of this dissertation.

Richard Swan deserves special thanks for designing, building and maintaining the hardware monitor which forms the backbone of this research. Professor Paul Shamon from the Statistics department was instrumental in the design of the instruction mix experiment. The instruction mix and many other experiments were made possible due to the generous help from Rick Fadden of Digital Equipment Corporation.

Credit also goes to my wife, Minakshi, who sacrificed many evenings and weekends to draw the figures and tables in this dissertation. Appreciation is due to Kamesh Ramakrishna and Navindra Jain who helped in getting the XGP output when I was editing the final draft of the thesis.

Final thanks go to the Computer Science Department at CMU which provided the stimulating environment and the sophisticated computer systems that made this research possible.

1. Introduction

1.1. *The Need for Performance Evaluation*

The effective measurement of computer systems is an essential part of the measurement, modelling and optimization cycle of computer systems. Computer systems have evolved into very complex structures, often consisting of many processing units, many I/O channels and a wide variety of high performance I/O devices. On the applications side, the computers are no longer operated by one programmer at a time executing a single small program. Today's computers are required to satisfy a wide array of demands from many users simultaneously. The users also expect the computer systems to obey (often conflicting) constraints of fast response and high component utilization, high throughput and low software overhead. It is no accident that many current computer systems are among the most complex man-made objects.

The increase in complexity and size is unfortunately not matched by an understanding of the dynamic behavior of such a system. This has given rise to the science (or art) of computer performance measurement and evaluation. The need to gain an understanding of the dynamic behavior of a computer system is felt by all computer professionals, whether engaged in design of new hardware and software systems or in writing efficient applications programs or in running a large installation. Lucas [LUCA71] has attempted to partition the need for performance measurement into three areas: design, purchase and optimization studies.

In the area of design and development of new hardware and software systems, measurement plays a vital role in guiding the designers to make optimal decisions with respect to design trade-offs. Moreover, many of the designs have to rely on analytical

or simulation models because no operational instance of such systems exists. Here, measurements are needed to obtain parameters for such models and to verify the predictions of the models.

The decisions regarding purchase or leasing of a hardware or software unit, can also benefit from performance measurement. The problem here is to compare the performance of the unit or system under consideration with some standard or with other systems. Since operational systems are available, in contrast with the design problem, direct measurement of parameters of interest is possible.

The third area of application of performance measurement is concerned with optimizing the operations of a specific computer system. The measurements are used to predict (and observe) the effects of small hardware or software changes. The changes include using a faster (or slower) central processor, acquiring faster or larger secondary storage devices, addition of primary memory, altering the allocation of devices to channels, altering the processor scheduling algorithm, altering the paging and other algorithms. The reason such optimization decisions cannot be made by the manufacturer once and for all is that the optimum choice depends on the workload experienced at an installation. The measurements have to be performed continuously and the system has to be altered (perhaps dynamically) to suit the changes in the workload.

We have chosen to partition the performance evaluation into levels such that a level is characterized by the number of machine cycles required for a typical operation at that level. It is advantageous to do so since the performance parameters, measurement tools and the applications of the evaluation are different from one such level to another. The lowest level is the hardware engineering level where the

operations involve one or a few machine cycles. The other levels in the increasing order are the hardware architecture level, the operating system kernel design level, the systems programming level, the applications programming level and finally the broad installation management level. These levels are discussed in more detail in the next chapter. It is not practical to attempt to investigate the performance aspects at all these levels at once since the parameters and the tools required are widely different. This dissertation, therefore focuses on the study of the performance measurement and analysis at the hardware architecture and the operating system kernel design levels.

1.2. Overview of the Dissertation

This chapter introduces the area of computer performance measurement and evaluation and discusses its applications. The next chapter presents the performance parameters at the various system levels and the measurement tools applicable to these levels. It also gives the motivation behind the research presented in the rest of the dissertation. Chapter 3 gives a brief description of our hardware monitor Kmon and discusses its strengths and weaknesses as they relate to the measurements at the hardware architecture and the operating system kernel design levels. Chapter 4 describes an in-depth experiment conducted to quantify the variation in the different application areas in terms of their usage of the PDP-11 instruction set. The complete statistical design used to quantify the variation is presented. Chapter 5 focuses on another major experiment relating to the operating system kernel design level for a multiprocessor system. The software contention arising due to mutually exclusive accesses to shared data structures is studied for Hydra - the operating system kernel for Cmmp (the Carnegie-Mellon multi-mini processor system). A simple central server

model is presented to study the contention. The model is validated against the measurements and then used to predict the lockout for larger C.mmp like structures. Chapter 6 presents the various experiments performed using our hardware monitor at other system levels. Some experiments are minor but some can become a major research project. Chapter 7 summarizes the dissertation with suggestions for future research.

Appendix A presents a survey of hardware monitoring techniques. A comparison of existing commercial and research hardware monitors is presented along many dimensions. Trends in the hardware monitor development are identified. Appendix B gives the complete instruction mix data which includes a short description of the measured programs. Appendices C through F present the outputs of our other experiments in detail.

2. Performance Parameters and Measurement Tools

Chapter 1 discusses the goals of performance evaluation and measurement. To achieve any of these goals, one has to perform experiments to measure many different quantities of interest. The purpose of this chapter is to identify the performance parameters of interest and to describe the various measurement tools in the context of these parameters. Even though our main interest lies in the areas of hardware architecture and operating system kernel design, we will present the parameters and tools applicable to other areas as well.

The literature on performance evaluation and measurement contains many instances of measurements of specific parameters. It is difficult , if not impossible to enumerate every parameter measured so far. Moreover, the architectures of existing hardware/ software systems and the emerging technologies give rise to interest in many parameters. For example, it is often no longer meaningful to measure the elapsed time to execute a program, but this was an interesting parameter before multiprogramming was introduced. Similarly, the advent of timesharing has generated interest in the response time measurements; and the emergence of multi-processors has given rise an to interest in parameters like memory contention and effective speed-up factor. Any list of performance parameters therefore stands to become obsolete as technology evolves. Just as performance parameters depend on the technology, the values measured for these parameters are also dependent on technology (that is, the characteristics of the components used to construct the systems and the way in which these components are interconnected) and the workload present on the machine when the measurements are made. These facts should be kept in mind while studying the performance parameters.

2.1. Classification of Performance Parameters

Various researchers have attempted to classify the performance parameters. Svobodova [SVOB76b] lists four different aspects of evaluation parameters-

1. Quantity of work a system can handle.
2. ability of a system to fulfil the users' needs (quality of service)
3. utilization of the system's hardware and software components
4. Internal operation and characteristics of the system's hardware and software components (underlying factors)

The first two are measures that describe a system as it manifests itself to an external observer. The last two describe the internal behavior of the system.

Fuller [STON75] notes that performance measures fall into two fundamental classes: response time measurements and throughput measurements. Measures in the class of response time include the time taken to respond to users' commands, time taken to service a disk request and turn-around time in a batch installation. The class of throughput measures includes the number of jobs executed per day and also the utilization of the various components.

We have chosen to classify the performance parameters as belonging to various levels in a computer system. This classification is advantageous because we believe that many measurement tools can best be classified as 'belonging' to the same levels. The idea of considering the performance parameters as belonging to different levels is not new. Kolanko [KOLA77] describes a scheme of considering levels of abstract machines, each composed of states that can be abstracted from many states from the

next lower level. A state transition at level h+1 (a state transition can be a measurement parameter) reflects any one of several transitions at level h. Svobodova [SVOB76b] considers the register transfer, the ISP, the software support and the PMS as meaningful levels for classification of performance parameters. Our classification (see figure 2.1) is similar, except that it considers the levels along the 'machine cycles' axis. We have also attempted to characterize the interest of various computer professionals along the same axis. Admittedly, such a classification is rather loose, but it does point out how different computer professionals view the performance parameters and measurement tools.

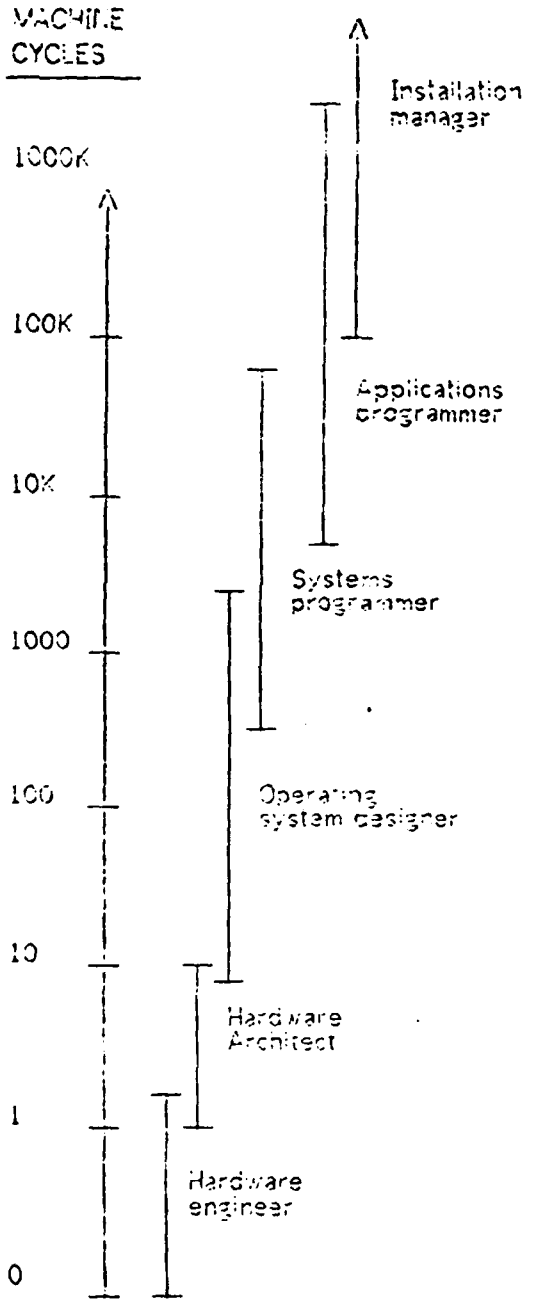


Figure 2.1 Performance Parameters

Relevant Performance Parameters	Most useful measurement tool
# of jobs per day, average cpu utilization, average channel utilization, device overlap	System accounting log
Execution profile, overlap structure, routine trace	Software monitor
Selection of resident and transient routines, file access methods (random or sequential) organization of files on disk execution profile size of various buffer areas	Software monitor
Resource allocation- cpu: think time, compute time, response time, length of queues, multiprogramming index core: working set size, core utilization, fragmentation, # of page faults per second devices: length of queue, distribution of request, time, service time, average wait time	Hybrid monitor
Time taken to execute individual instructions, instruction mix, addressing modes, register life, use. Distance between jumps, branch probability, Locality, Cache hit/miss	Hardware monitor
Bus bandwidth, # of read, write cycles. Effect of multiprocessor memory contention, I/O cycle stealing.	Oscilloscope

2.2. Performance Measurement Tools

Figure 2.1 points out that there are a variety of performance measurement tools , with each level having a prefered measurement tool. For an installation manager, the goals of performance measurement are to aid the purchase of new components or systems and to direct the optimization of existing resources. To achieve these goals, two main measurements have to be performed: the workload at the installation has to be characterized and the utilization of various hardware components needs to be measured. The system accounting log gives the resources consumed by individual user jobs and so it can be used for workload characterization. Moreover, with a more sophisticated log, one can get the resource utilization on a per-second basis. The measurement of overlap of various hardware components however, has to be obtained using a hardware monitor in present computer systems.

The reason a software monitor is most applicable at the applications and systems programming levels is two-fold. First, measurements at these levels need considerable amount of structured information in the form of various system queues, job tables and the association of high level language statements with actual instructions being executed. Such information is most easily obtained by introducing measurement code in the appropriate routines. Second, the primitives at these levels take many thousands of machine cycles to execute, so the overhead caused by inserting measurement code is not prohibitive.

As one proceeds to lower levels, the overhead caused by software monitoring becomes significant and moreover, the high level information needed to gather or interpret the measurements can usually be compressed into a few bits (e.g. whether a

certain user is executing, operating system or user execution). It is therefore natural to use a hybrid monitor where most of the measurements are performed using the hardware monitor, but the software on the measured system assists by supplying the required high level information.

Finally, when one is considering events taking place over a few machine cycles, the only tool that can be used is a hardware monitor. The machine states required to be monitored at this level are generally not accessible via software. Measurements at the cycle level and below are usually conducted by a hardware maintenance engineer and they are usually performed for the purpose of diagnosis rather than performance evaluation. We will not be concerned with measurements at this level.

The need for measurements at the hardware bits level gave rise to the hardware monitors. In fact, most of the early studies involving hardware monitors were restricted to this level [ASCH71, BOND71, IBM63, RUDD72, TAIB72]. Clever ways have been devised however, to make a hardware monitor useful at upper levels. A simple address comparator can be used in some systems to identify that a particular user is executing or that a particular operating system function is being performed. A number of performance parameters applicable to higher system levels can be measured using a hardware monitor without altering the software on the measured system at all (e.g. the average CPU utilization, the execution profile of the operating system, average think time and compute time). When the software on the measured system is modified to actively assist the hardware monitor, one can perform measurements on the activity of a particular user or take into account the effects of program overlays when acquiring an execution profile or a routine trace. Such a hybrid measurement technique has been found to be quite useful [SVOB76b, HUGH74, COLL76, SEBA74]

Even though figure 2.1 states that the most applicable tool at the level of the installation manager and applications programmer is a software monitor, we find that most commercial hardware monitors, by virtue of the software supplied with them are geared towards these upper levels. Apart from there being more market at these levels, we see the following reasons:

1. Parameters at upper levels are composed of parameters at lower levels and these can be monitored using a hardware monitor. The device overlap and device utilization certainly need a hardware monitor in current systems. Moreover, parameters like the average response time actually consist of lower level parameters like the execution of command interpreter, device initiation to bring in a new program if required, wait for the device to complete the request and execution of the new program. Each of these can be measured using a hybrid or a hardware monitor. Even when such measurements are unable to give a concrete number for the average response time, they indicate why the observed response time is large, so corrective action can be taken. In short, lower level measurements can not only yield upper level parameters, but in some cases can provide valuable insight.
2. Any software monitor necessarily implies some modification of existing the operating system , systems programs or user programs (a sampling software monitor is an exception, but its applicability and accuracy is limited if the sampling rate is kept low to reduce overhead). Such a modification perturbs the system and more importantly, can become a source of errors. This gives rise to the reluctance in gathering information with a software monitor if the same can be obtained using a hardware monitor.

3. A hardware monitor, once set up, can give quick answers to many questions. For example, parameters like the average time taken to execute a certain routine or the distribution of use of different supervisory service calls and the time taken to complete each call can be obtained without too much effort. Equivalent measurement in software would require more time and a more detailed knowledge of the software under consideration.

2.3. Performance Parameters Constituting Our Study

We are interested in the performance measurement and evaluation at the hardware architecture and operating system kernel design levels. These fields are expanding rapidly with the advent of technology and it is therefore not appropriate to attempt to study all aspects of performance related to these areas in depth. We have studied many parameters of interest in these areas and have extended our study to include multiprocessor systems as well. Our viewpoint is always that of the designers attempting to study the hardware architecture and the operating system kernel by measuring the behavior of the existing systems under actual user programs. Figure 2.2 displays the major parameters constituting our study.

Figure 2.2 Performance Parameters Constituting our Study

	Uniprocessor systems: behavior of systems and workload characterization	Multiprocessor systems: contention due to shared resources
Hardware architecture level	The instruction mix and quantification of its variability with respect to application areas (chapter 4)	Memory contention in C.mmp (chapter 6)
Operating system kernel design level	Study of system behavior and workload characterization is made difficult by non-uniformity of primitive functions across operating systems	Software contention in Hydra for a non-interactive workload (chapter 5)

For the hardware architect, the variability in the workload has always been a difficult design problem. It is intuitively clear that different application areas (scientific, business, process control) present different workloads to a processor. The important questions faced by the designer in this respect are: are the application areas different at the lowest level of data structure manipulations i.e. the instruction mix level? If so, are they sufficiently different to justify a specialized processor for each application area? How much performance improvement can be obtained by such specialized processors across all the programs in a given area? We apply statistical experimental design techniques to quantify the variance in the instruction mix due to the various factors in order to answer these questions.

The measurement of the workload and its variation for the operating system kernel design level is complicated by the fact that each operating system kernel has its own set of primitive functions and comparisons across different operating systems is not possible with our current understanding of operating systems. We will therefore not study parameters at this level for uniprocessors.

In the case of multiprocessors, the study of the contention for shared resources among processors is very important. At the hardware architecture level, the contention occurs for the shared memory and shared data paths if any. This problem has been studied earlier by others ([BHAN73], [MCCR73], [BASK76]) using analytical models. We have attempted to measure the memory contention for C.mmp - the Carnegie-Mellon University's multi *miniprocessor*. Our study was hampered by the lack of high resolution measurement tools. However, the contention problem at the kernel design level was attacked successfully. The contention arises because in order to maintain integrity in a multiprocessor system, certain shared data objects (such as the list of runnable processes or the list of free blocks of memory) have to be accessed by only one *processor* at a time. When two or more processors attempt to access the same shared data object at the same time, only one of them can access it and others have to wait. The mechanism used for such mutual exclusion is called a lock. The time lost by a processor while waiting for a shared object to become free can become a performance bottleneck for multiprocessors. A hardware monitor was used to measure the locking behavior and the contention occuring in Hydra, the operating system for C.mmp. To the best of our knowledge an experimental investigation of this problem was not possible in the past. Our study of this important performance parameter should guide the study of this problem in future multi-processor operating systems.

It is clear from the discussion in the previous section that a hardware monitor is the most appropriate tool to investigate these parameters. But a hardware monitor is a versatile tool which can also be applied to performance studies of other system levels. We have therefore expended some effort in studying the hardware monitoring techniques in general. The next chapter briefly describes our hardware monitor K.mon and in chapter 6 we examine how it has been used for measurements relating to the different system levels.

3. Description of K.mon

3.1. Introduction

K.mon is a memory bus monitor for the PDP-11 family of computers capable of monitoring every cycle taking place on the PDP-11 Unibus [DEC71]. Since most computer system components and peripherals on a PDP-11 communicate with each other via the Unibus, the Unibus is a rich source of information regarding every activity taking place in the computer system. In principle, it is possible to record every cycle occuring on the Unibus and post-process the data to obtain the required information regarding any activity on the system. It is however impractical to record all the cycles at the rate they occur. Moreover, the post-processing program will have to be very complex to extract the required information from a large Unibus cycle trace. K.mon therefore provides very sophisticated event detection mechanisms which enable the analyst to record only the events of interest thereby simplifying the task of recording and post-processing. K.mon gives access to hardware level performance information not otherwise available. Moreover, the event detection mechanism is capable of obtaining information regarding software performance without the insertion of software breakpoints.

This chapter is intended to describe K.mon to a level of detail necessary to understand the experiments presented in later chapters. The motivation for K.mon and its design philosophy is discussed by Fuller, Swan and Wulf [FULL73]. For a more detailed description of the operations of K.mon, the reader is referred to [SWAN76].

3.2. The Experimental Setup

K.mon presents itself as a passive device on the Unibus of the processor under measurement (P.host). Another processor (P.sup for supervisor) is required to control K.mon and to store the data gathered by it. K.mon thus straddles two Unibuses as shown in Figure 3.1. The figure shows K.mon connected to C.mmp such that both P.host and P.sup are processors on C.mmp. It should be noted however, that either or both of the processors may be conventional stand-alone PDP-11's. In fact, this was the mode in which K.mon was used for most of our experiments[1].

All the signals present on the P.host Unibus are available to K.mon. In addition, a set of 24 probes is provided to monitor signals not available on the Unibus. The probes are currently used to monitor the following signals:

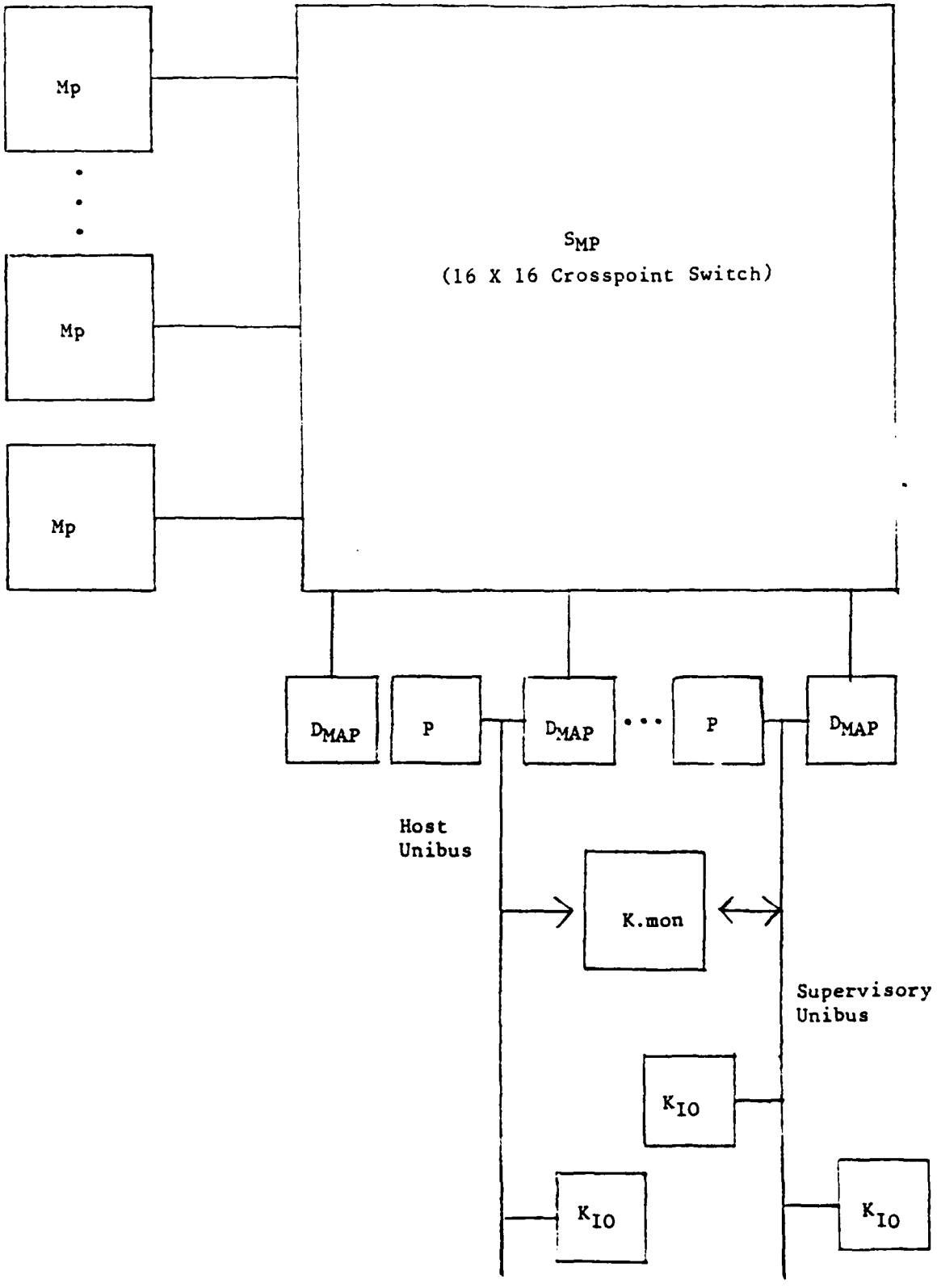
1. the instruction fetch signal used to distinguish between the instruction fetch cycles and operand, data or I/O cycles.
2. a signal indicating that a cycle is initiated by a processor as against initiated by an I/O device.
3. three bits giving the priority level at which the processor is running.

The input signals are tested combinatorially at the occurrence of each Unibus cycle on P.host to detect events of interest. K.mon can be programmed to record information such as the address or data involved in the cycle when an event occurs.

---

[1] There are many advantages in having both P.host and P.sup to be processors on C.mmp. It makes the full resources of C.mmp available for P.sup to store and post-process the data. More importantly, the data obtained by K.mon can be used by the operating system for dynamic tuning. The debugging capabilities of K.mon are also enhanced by providing assistance from the operating system.

Figure 3.1 K.mon connected to Cmmp



The information is recorded in a buffer (80 events deep) and is then transferred to the main memory of P.sup via a standard interface (DEC DR11-B).

P.sup controls the K.mon via five command registers which are used for initialization, setting of the operating modes and reporting exceptional conditions. The event detection and event response functions in K.mon are completely programmable and they are determined by a 65 word specification word memory (SWM) which is set up by P.sup prior to running an experiment. Figure 3.2 shows the block diagram of K.mon and its relationship to the two processors.

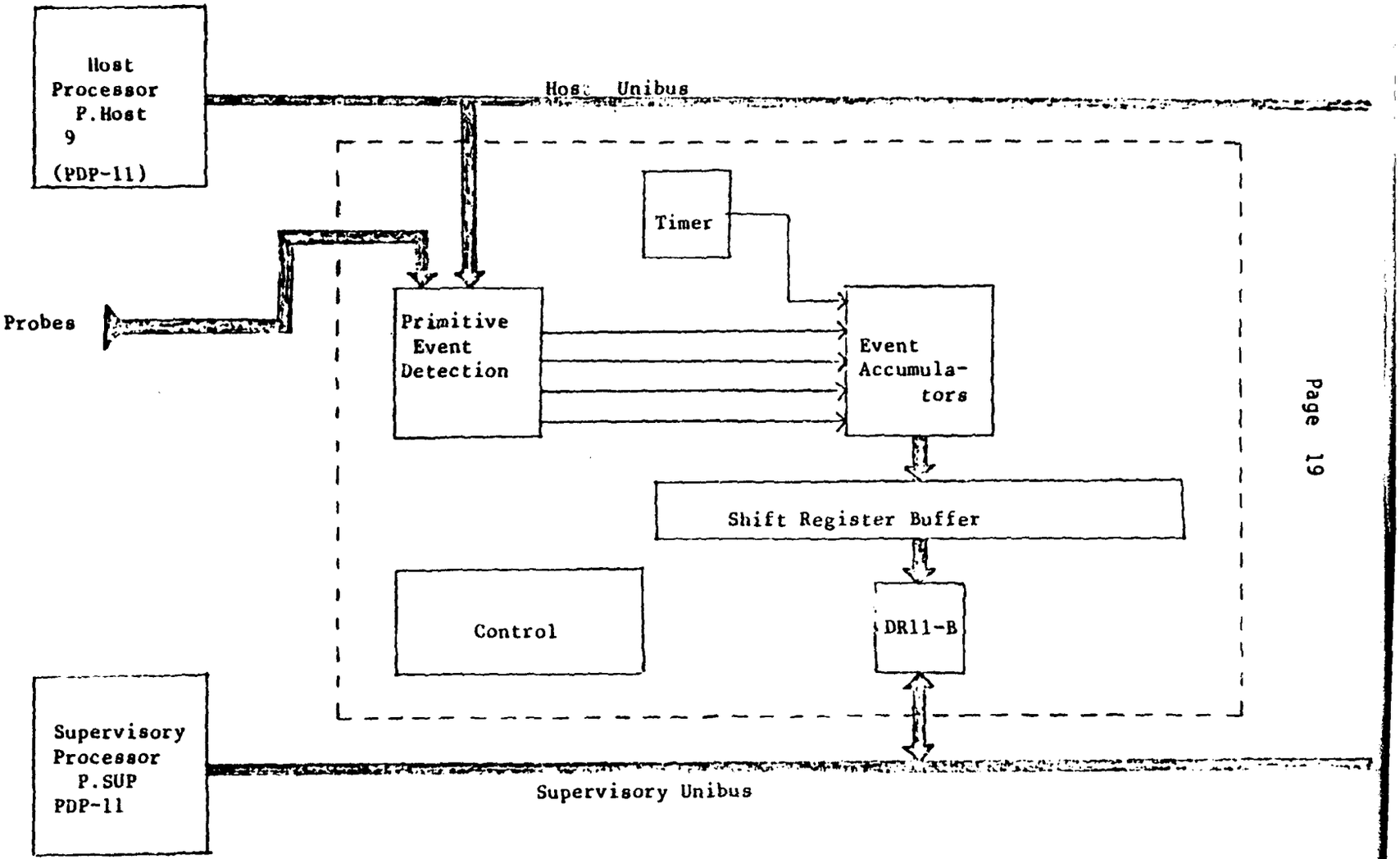


Figure 3.2 K.mon Block Diagram

3.3. Event Detection

The concept of an event is central to all hardware monitors. An event can be loosely defined as an occurrence of a particular state on the system under measurement. The event we are interested in can also be a combination or a sequence of other events. An event can be as simple as an occurrence of an instruction fetch cycle or as complex as the occurrence of the first operand fetch cycle after executing the instruction at a certain location while executing a particular user's program.

Since events can be so complex and since different events need to be monitored for different experiments, the event detection mechanism is the most important part of a hardware monitor. In K.mon, events are detected at two levels- primitive and accumulated. A primitive event can occur on every Unibus cycle. The primitive events are counted until a specified number of them happen leading to an accumulated event. The accumulated events and K.mon's response to them are discussed later in this chapter.

A primitive event is the lowest level of resolution of the K.mon. During each cycle on the P.host Unibus, all the available signals are latched. The input signals are inspected simultaneously by four distinct combinatorial logic units to detect four distinct primitive events. The input signals are divided into four different groups for the purpose of detecting sub-events which are combined to detect a primitive event. The groups of input signals are:

1. 16 bits of Unibus address
2. 16 bits of Unibus data
3. 16 bits of probe signals or 8 bits each of probe signals and Unibus cycle length
4. 7 bits of control signals:

2 bits: Unibus address bits 17 and 18
2 bits: cycle control signals:
read, read-pause, write and write-byte
1 bit: signal indicating that the cycle is
an interrupt request cycle
2 bits: internal flags used for detecting sequences

The sub-events are detected using two *functional units: comparators and pattern* detectors. A comparator performs a 16 bit unsigned arithmetic comparison between its internal comparison value register and an external signal group (usually the 16 bit Unibus address). The two result signals are 'Equal' and 'GEQ', indicating that the input signal is equal to, or not less than the comparion value register. A pattern detector is used with the data, probe and control signal groups to detect any particular bit pattern. It consists of two internal registers: mask and pattern. The mask register is used to identify the care/ don't care bits of the input signal. The result signal 'Match' is true iff

(inout signal ∧ mask register) = (pattern register ∧ mask register)

3.3.1 Combination of the subevents

Figure 3.3 displays the arrangement of the comparators and the pattern detectors. For each primitive event there are four sub-events -address match/GEQ, data match, probes match and control signals match. Each of these sub-events can be tested for being true or false or can be ignored in defining the corresponding primitive event. The right half of the control bits pattern detector is used to perform this final primitive event detection function using the same pattern and mask register concept. Even though only one event comparator is provided for each primitive event, it is possible to specify a primitive event which inspects the Unibus address for being in a certain range by using the GEQ signals from two comparators.

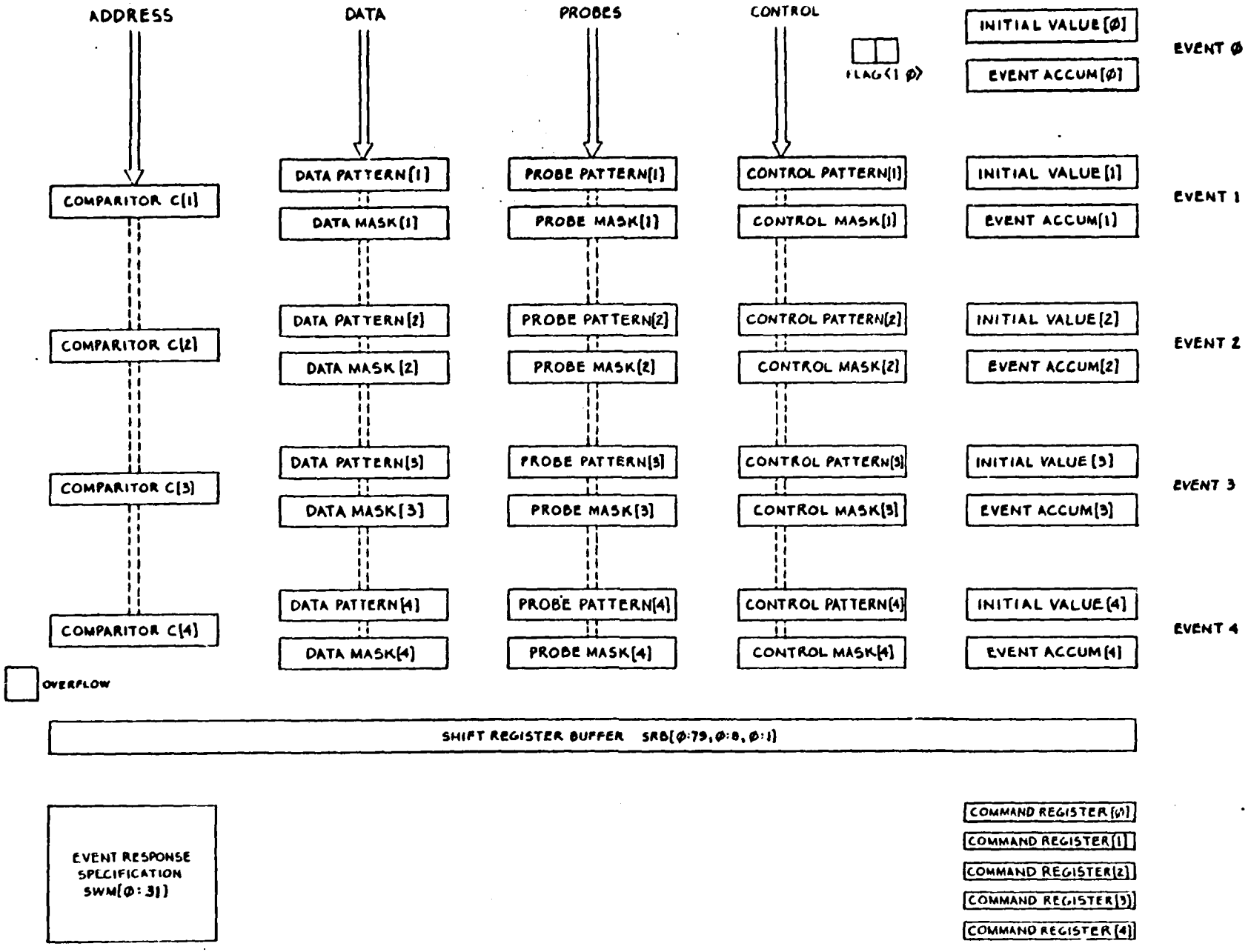


Figure 3.3 K.mon's Internal Registers

3.3.2 Accumulated Events

It is sometimes necessary to determine how many times a certain primitive event happens between two other events. The event accumulators provide the required counting function for this purpose. There are four event accumulators, one for each primitive event. An event accumulator consists of a 16 bit counter and a 16 bit initial value register. When a primitive event happens, the counter in its event accumulator is decremented by 1. When the counter reaches zero, an accumulated event is said to have happened. K.mon responds to an accumulated event by recording certain information as described in the next section. In addition, the counter is loaded with the contents of the initial value register to enable subsequent count-down. Note that the initial value register can be set to zero causing an accumulated event every time the corresponding primitive event happens. Another commonly used value in the initial value register is $2^{16}-1$ which is the maximum value it can be set to. The event accumulator counter then overflows infrequently and can be used to count the number of times the corresponding primitive event has happened. For example, suppose the initial value register for primitive event 1 is set to its maximum value. By recording the value of this counter when other events happen, one can determine how many times primitive event 1 happened between any other events.

This mechanism is also used for determining the time elapsed between two events. A special event accumulator is provided for this purpose which counts the occurrences of a clock tick. In other words, the primitive event for it happens at a constant rate (programmable to be 1 , 16, 256 or 4096 microseconds).

3.4. Event Response

When an event is detected, it is sometimes sufficient to just record its occurrence whereas at other times, it is necessary to record more information like the address or the data that caused the event to take place. A time stamp and the values of internal event accumulator counters may also be required for later analysis. In the early monitors, only summary type information was made available. So the only response to any event was to increment an internal counter. This is sufficient if only gross average values of the measured quantities are required. If however, one needs to generate histograms for constructing analytical or simulation models, more detailed information has to be obtained by the hardware monitor.

In K.mon, since there are five event accumulators, any combination of up to five accumulated events can happen simultaneously. In some experiments, when two accumulated events happen simultaneously, different action needs to be taken than if they happen separately. K.mon therefore provides for an event response specification word for each of the 31 combinations of the five accumulated events. When an event is detected, up to 9 words of information can be obtained. These are: address, data, probes, miscellaneous signals, clock value and four words giving the number of times each of the four events has occured so far (i.e. the counters in the four event accumulators). Moreover, two internal flags can be set or reset to facilitate detecting a sequence of events. If a special timer mode is set, flag 0 can be used to enable the timer. This provides a dynamic event driven mechanism to start and stop the timer. The miscellaneous signals word contains the flag values, the Unibus control signals and an identification of which of the accumulated event(s) happened.

In some experiments, the rate at which the data is gathered exceeds the rate at which it can be transfered on the P.sup Unibus. Two buffers each with 80 events capacity are used in a double-buffering mode to smooth the flow of data to P.sup. When the buffers overflow, P.sup is interrupted to take the necessary action such as disabling event detection or reinitializing its programs. When an overflow happens no further data can be gathered until the buffers become free again. Information about events happening during the overflow condition cannot be obtained.

3.5. Strengths and Weaknesses of K.mon

As we shall see in the following chapters, K.mon has been successfully used for the study of instruction mixes and the software lockout phenomenon. There are many other measurements for which K.mon is not an appropriate tool e.g. memory switch contention or tracing of all cycles taking place in the system. The amount of measurement space spanned by a monitor can be loosely defined as the power of the monitor. Of course, in addition to power, the ease of use and the ease of attachment to the measured system are also important parameters of a measurement tool. Svobodova [SVOB76b] defines monitor power as composed of four components: monitor domain, monitor rate, input width and recording width. For the purpose of discussing K.mon, we have partitioned the monitor power into the following four dimensions:

1. The monitor domain of a hardware monitor consists of the signals monitored with the help of probes. In the case of K.mon, the domain consists of the 18 bits of memory address, the 16 bits of memory data, 5 bits of control signals and 5 bits of special signals making a total of 44 bits.
2. The monitor rate is the rate with which events can be detected - limited by the monitor's probes and logic.

3. The input width of a monitor is the total number of bit comparisons against the input signals, that can be performed for the purpose of defining events in an experiment. For K.mon, the input width is 4 primitive events times the 44 bits of its domain.

4. The recording rate of a monitor is the maximum possible rate (in bits/sec) of sustained output to the supervisory processor or some secondary storage device. This dimension of power is a crucial parameter for experiments involving tracing. For measurements involving counting and sampling, the recording rate can usually be ignored. For K.mon, the recording rate is limited by the rate at which it can transfer data on the supervisory processor's Unibus.

3.5.1 The Monitor Domain

K.mon is a memory bus monitor, that is, most of the signals in its domain are already available in one place on the memory bus (the Unibus). Most commercial monitors are designed for monitoring many different computers and so their domain has to be established using high impedance probes to pins in the host computer. This approach has many drawbacks (see [SVOB76b] chapter 5) such as inaccessibility of pins, danger of causing a hardware failure and using a wrong pin. Our experience has indicated that monitoring the memory bus is sufficient for most experiments and moreover, in most cases, the software on the host computer does not have to be modified at all. A special monitor register has been proposed by Hughes ([HUGH73]) which is controllable by P.host software using special instructions. This register is monitored by a hardware monitor to gather the information supplied by the software. A memory bus monitor like K.mon eliminates the need for such a register since any location in the main

memory can act like this register - the hardware monitor needs only to monitor accesses to this location. Our experience with RSX11M hooks (see section 6.2.3) clearly shows the advantages of a memory bus monitor.

It is important to monitor address lines before any address mapping is performed, so that address seen by the monitor is the virtual address generated by the program under measurement and not the absolute memory location address. By doing this, the measurement is not affected by any dynamic relocation of the program. The fact that all the peripheral devices are controlled via Unibus registers expands the domain of K.mon considerably. K.mon's address comparators can be used to monitor the commands being given to the peripheral devices.

Many of the measurements performed by K.mon can be performed in a microprogrammable host by suitably altering its microcode. We will refer to this technique as firmware monitoring in the rest of this chapter. Such a firmware monitor has a larger domain (e.g. internal registers) but it lacks the ability to monitor events inside the devices and the also transfers between the devices and memory. It should be noted that a hardware or firmware monitor has a very restricted domain compared to a software monitor. A software monitor essentially has all memory locations which can be read using an instruction as part of its domain. It cannot however monitor device activity unless the CPU is involved in it.

An entirely new problem arises when one is monitoring a multi-processor or a network of computers. The domain of a monitor needs to be expanded to include all the processors in order to study the operation of the system as a whole. Kolanko [KOLA77] and Tesdata report using a hardware monitor to measure a network of computers. Our experience with C.mmp suggests that hardware monitoring of a multi-processor faces two main problems:

a. The monitor has to accommodate a larger domain where different signals in the domain are valid at different time instants

b. The monitor should be able to handle a higher input rate

Similar problems arise when using multi-port memory or when the processor uses different buses for communicating with different memories.

3.5.2 The Monitor Rate

The fact that K.mon is a memory bus monitor, sets the maximum useful monitor rate to be the rate at which memory cycles occur on the host memory bus. This arrangement forbids any measurements at levels below a memory cycle (e.g. cache hits, length of a cycle [2]), but it was used to simplify the synchronization problems. The high event detection rate of hardware monitors has traditionally resulted in their use for certain counting type measurements. In fact, since the recording rate of a hardware monitor is usually low, whenever events are to be detected at a high rate, the most common response to an event is incrementing a counter depending on the event occured. Svobodova [SVOB76b] has proposed hardware aids for internal monitoring. These include counters for counting events commonly counted by a hardware monitor (e.g. memory cycles, instructions, channel use and overlap, various timers). If processors are designed with such internal hardware aids, many experiments can be performed using software monitors without the need of an expensive hardware monitor.

---

[2] K.mon has to employ separate circuits for measuring the length of a cycle

3.5.3 The Input Width

The input width of a monitor determines the number of different events that can be defined in an experiment. The input width of K.mon is fixed at 4 times 44 bits. This number is somewhat inflated, however, since for most experiments only a part of this total width is used. For monitors which employ a manual patch board for event detection, the input width is variable and each experiment is typically defined with the minimum required width. This approach results in a saving of hardware components (e.g. comparators, bit masks) at the expense of the ease of use and the time required to set up an experiment. The loss of flexibility resulting from the loss of a manual patch board did not restrict the applicability of K.mon.

For multi-processors and computer networks, the input width of a monitor has to be increased since the number of bits needed for an effective study increases. It was originally proposed to build one K.mon for each processor in C.mmp. This solution to the input width problem cannot usually be adopted because of the expense involved.

3.5.4 The Recording Rate

The definition of the recording rate assumes that the hardware monitor has access to some high speed secondary storage device either directly or through a supervisory computer. A high recording rate is highly desirable since it lets an analyst take full advantage of the high input rate of the hardware monitor. The experiments performed using K.mon indicate that the ability to use the hardware monitor in tracing mode is very valuable. In the tracing mode, the monitor acts like a filter allowing selected Unibus cycles to be recorded. The traced data is then post-processed to generate tables and histograms. There is a trend in hardware monitors to perform data

processing and storage operations within its own hardware. In these monitors a special tracing mode needs to be provided.

The recording rate is affected by many factors. In the hardware monitor itself, high speed buffers (at least two) need to be provided so bursts of data can be recorded without any loss. The high speed buffers need to be emptied into the main memory of P.sup (if one is used). The final stage is storing the data on a magnetic tape or a disk. A bottleneck can be present at any of these transfers, which results in a diminished recording rate. In the case of K.mon, the buffered data inside K.mon is transferred to P.sup via its Unibus. The recording rate is therefore limited by the speed of P.sup's Unibus.

4. The Instruction Mix Experiment

4.1. Introduction

In the last two chapters we examined the performance parameters at the various system levels and the hardware monitoring techniques employed to study these parameters. This chapter describes a major experiment designed to answer some important questions regarding the architecture of a computer.

One of the most important set of measurements at the instruction level is the instruction usage i.e. the instruction mix. The instruction mix, combined with the average time taken to execute each of the individual instructions, yields the average overall instruction execution time for most straightforward processor implementations. This is a measure of the raw speed of a computer and so it can be used in comparing different computers that share the same architecture. It should be noted however, that in the present generation of computers, the effects of pipeline architecture and cache memory reduce the importance of the instruction mix in the determination of the average overall instruction execution time. The main use of instruction mixes is now in the design and implementation of the instruction set processors.

Another important parameter at this level is the usage of addressing modes and special registers. Earlier computers had special index registers in addition to general purpose registers. The use of index registers considerably speeds up address calculations for accessing arrays and other data structures. Similarly, in some computers, a special 'indirect' bit in the instruction word indicated that the address provided is actually a pointer to the real address. The frequency of use of index registers was reported by Gibson [GIBS70] as a separate instruction in the 'indexing'

class because of its importance. Our study is based on the instruction mix for PDP-11 for which there are no explicit index registers or indirect bits. Instead, it has 8 addressing modes which are used with any of the 8 general purpose registers as follows [DEC71]:

Table 4.1 PDP-11 Addressing Modes

Mode	Symbolic	Description
0	R	operand is in R
1	(R)	address of the operand is in R
2	(R)+	same as mode 1; but R is incremented after use
3	@(R)+	address of the address of the operand is in R; R is incremented after use.
4	-(R)	same as mode 1; but R is decremented before use
5	@-(R)	same as mode 3; but R is decremented before use
6	X(R)	X+ value in R is the address of the operand
7	@X(R)	X+ value in R is the address of the address of the operand

Registers 6 and 7 have specialized functions. Register 6 is used as a stack pointer and register 7 is the program counter. Register 6 is usually used with modes 2 (pop), 4(push) and 6(parameter access within the stack). Register 7 is usually used with modes 2(immediate), 3(absolute) and 6(relative). Some instructions have only one operand (e.g. INCrement, CLR), which is specified by a single mode-register pair. There are also double operand instructions (e.g. MOVe, ADD, BITtest) which have a source and a destination operand each specified using a mode-register pair.

4.1.1 Uses of Instruction Mix

Even though, historically, the instruction mix has been used to calculate the average instruction execution speed of a computer, it has found many more uses. Among these are:

1. Design of future processors.

   Design of a new instruction set involves trade-offs between cost of implementation and power of the instruction set, between hardwired implementation and microcoded or software implementation, between opcode encoding and time needed for opcode decoding and so on. Moreover, decisions have to be made regarding provisions for immediate operands, prefetch of instructions and operands and number of general purpose registers. All these decisions need the instruction mix data to make optimum choices.

2. Emulation of an instruction set

   Due to the considerable software effort vested in the existing instruction sets, it is advantageous if new processors can emulate instruction sets of their predecessors. The efficiency of emulation can be increased by providing special features in the hardware of the new processor. Some of the emulation can be performed using microcode, some can be done in software without too much time penalty. Instruction mix helps in deciding the level at which an instruction can be emulated.

3. Designing a special purpose implementation of an existing instruction set

   The instruction mix data can sometimes indicate that a particular application is

strongly biased in favor of using certain instructions [3]. In such cases, a special purpose implementation (in the form of a new model or a hardware option) of the instruction set optimized for the specific instructions will be useful. The advantage to be gained by doing this can be quantified using the instruction mix.

4.2. Review of Previous Work

Studies of frequency counts for instruction executions have been described by several authors. The best known is the Gibson mix (see [GIBS70]), developed by Jack C. Gibson at IBM in 1959. Gonter [GONT69] has compared the Gibson mix with the University of Massachusett's mix. His results correlate well with Gibson's. The substance of these results is that LOAD and STORE account for about 30% of the instructions executed, branches for 16% to 38%, index manipulations 13% to 18% and arithmetic 3% to 19%. These results depend on both the ISP and the subject program set. Other similar mixes are reported by Arbuckle[ARBU66], Conners, Mercer and Sorlini[CONN70], Raichelson and Collins [RAIC66]. Foster, Gonter and Riseman [FOST71] have gone one step further, by investigating the effects of reducing the instruction set. The emphasis of the above studies was mostly on the evaluation of raw processing capacity of the central processor. The subject programs were limited to user programs. Lunde [LUND74] measured the instruction mix for PDP-10 and also studied the register utilization and commonly occuring instruction sequences.

---

[3] In fact, the variation in the instruction mix from application to application forms the basis of this chapter.

4.2.1 Methods for Obtaining Instruction Mix

A variety of methods have been used by researchers to obtain the instruction mix.

1. Instruction or machine cycle tracing

   Earlier mixes were measured using software tracing. In this method, every instruction (or machine cycle) is recorded to obtain the instruction (and operand) values and other relevant information. If a hardware monitor is used for tracing, operating system execution as well as user program execution can be traced. The problem here is that the internal memory of the hardware monitor gets filled rapidly and it cannot be emptied into secondary memory fast enough to avoid overflows [BORD71]. Software methods rely on interpretation of user programs [LUND74, WIND73], with an interpreter designed to gather required statistics. The disadvantages of this method are that only user programs can be traced and moreover, the execution of the traced user programs gets slowed down by orders of magnitude. This method therefore cannot be used to obtain the instruction mix for real-time programs. Even with these disadvantages, tracing has been used because such a trace is a rich source of information. Apart from calculating the instruction mix, the trace can be used to gather information on register lives, sequences of opcodes, address calculation, locality of reference and distance between branches. An instruction trace can also be used to drive a processor simulator to evaluate various paging and other algorithms.

2. Microcoded measurement facility

   Recently [SVOB76a] microprogrammed processors have been used to gather the

instruction mix data for the instruction set being implemented on such a processor. Along with interpreting the instruction set, the microcode was programmed to collect the instruction mix in the fast storage available internal to the microprocessor. This method is similar to the previous method but it introduces very little overhead and is certainly cheaper than employing a hardware monitor. It is however, applicable only to microprogrammed processors.

3. Jump trace

Alexander [ALEX72] describes a variation of tracing called Jump tracing. In this method, tracing information is gathered only when the flow of control is altered. This method introduces less slow-down for the user program but the trace produced is not as useful as the complete trace. Moreover, to do this entirely in software requires the compiler to insert appropriate code to activate the tracer at the proper jump points.

4. Sampling

When detailed information is not required, one can obtain parameters like the instruction mix or execution profile by sampling the processor state at random. Software samplers are time driven and interrupt the processor at random times to obtain the instruction or program counter in use at the time of the interrupt. Software samplers therefore cannot sample uninterruptable operating system code. Moreover, separate samplers have to be written for every operating system. A hardware sampling monitor removes these restrictions. Also, such a monitor does not cause any overhead or perturbation in the operation of the system under measurement. We have therefore chosen this approach for our study.

4.2.2 Indication of Variation in the Instruction Mix

Most of the researchers measuring the instruction mix have reported that the mix is dependent on the application area chosen for measurement. Let us briefly follow through the various studies observing the variation in the instruction mix. On the highest level, there is variation between different processors (i.e. between different instruction sets). Some of this variation is of course due to the processors being intended for different application areas and due to non-uniformity of opcode definition from processor to processor. It is however, instructive to group opcodes in certain groups for comparing different instruction sets as was done by Gibson. The following table is reproduced from Lunde.

Table 4.2 The modified Gibson classification

Percentage of the executed instructions in the Gibson classes

Instruction class	Gibson's results IBM 650/704	UMASS results CDC 3600	Lunde's results PDP-10
Load,store	31.2	30.0	42.4
Integer add,subtract	6.1	1.2	12.4
Compares	3.8	1.2	----
Branches	16.6	38.3	28.2
Floating add,subtract	6.9	0.5	4.9
Floating multiply	3.8	0.5	2.6
Floating divide	1.5	0.2	1.1
Integer Multiply	0.6	0.1	1.1
Integer Divide	0.2	0.1	0.5
Shifting	4.4	2.2	3.9
Logical	1.6	0.5	1.0
Miscellaneous	5.3	0.0	1.5

Svobodova [SVOB74] has compiled a set of instruction mixes for the IBM 360/370 series of processors. These processors have very similar instruction sets. The variation in the instruction mix results from measurements performed at different installations. The table given by Svobodova is reproduced below:

Table 4.3 Instruction Mix at Different Installations

	Installation/ Model			
Opcode type	Stanford University 370/145	Argonne National Laboratory 360/75	UCLA 360/91	RCA Laboratories 70/45
Integer load, store and arithmetic	26.51	50.85	25.21	25.7
Floating- point load, store and arithmetic	0.52	2.82	28.62	-
Decimal	0.06	-	-	5.0
Branch	32.74	26.04	18.30	34.2
Logical, compare, move	18.97	17.15	13.41	17.1
Control, I/O	0.54	0.37	-	-

Note: IBM 370/145, 360/75 are arithmetically oriented.
IBM 360/91 has a powerful floating point unit
RCA 70/45 has an instruction set almost identical to that of the IBM 360.
The instruction mix was obtained from tracing user programs (mostly Cobol)
- means no information was available in the reference

Hughes [HUGH76a] reports variation in the instruction mix between user programs and the operating system. Finally, Svobodova and Mattson [SVOB76] report that the instruction mix can vary between different phases of execution of a single program.

4.3. Statement of the Problem

It can be seen from the above discussion that there is noticeable variability in the instruction mix from machine to machine, from installation to installation for the same machine, from program to program in the same installation and even from phase to phase in the same program. For programs written in assembly languages, the instruction mix will depend on the particular programmer writing the program. This makes the use of the instruction mix for design and optimization of processors somewhat difficult. Any measurement of the instruction mix which does not span all areas of application of the measured processor, cannot be assumed to represent the overall instruction mix experienced by the processor. Since the extent of variation in the instruction mix is not known, the processor designer faces the following pitfalls:

1. If the variation in the instruction mix is significant, the cost/performance ratio degrades if a non-representative mix is measured and more attention is given to unimportant instructions at the expense of important instructions.
2. A processor optimized for a balanced instruction mix is suspected of being unoptimized for a particular application area and is therefore not used even though the actual variance in the instruction mix from area to area is negligible.

It is important to do a scientific study to quantify the variance caused by the different factors in the overall instruction mix. Quantification of the variance assists the hardware architect in avoiding the above pitfalls. The variation in the instruction mix is

caused by many factors, some of which were discussed in section 4.2.2. We list below the most important factors that have an impact on the instruction mix:

1. The instruction set of the processor
2. The broad application area
3. The individual programs belonging to the different application areas
4. The different phases of execution in a program
5. The compiler used to translate the high level program into the machine language
6. The individual programmer in the case of assembly language programs

We have decided not to investigate the variance caused by the differences in the instruction set and by the individual programmers since these will form complete studies by themselves. The methods used in our study can however be extended to study the variance caused by these two factors also. We will also not study the variance caused by the use of different compilers mainly because if such a study is not to become too compiler specific, it needs many different compilers written for the same language and for the same processor. This was not possible even for the most popular languages viz. Fortran and Cobol.

The goal of our experiment is to compare the instruction mix for different application areas, programs and execution phases in the programs to quantify the variance caused by each of these factors. If a certain broad application area (business, real time) is found to have a significantly different instruction mix, it might be worthwhile to design/ implement a special processor or option for that area. But even in a single application area, all the programs cannot be expected to exhibit the same instruction mix. If the variance due to individual programs is found to be comparable to the variance caused by application areas, it will not be possible to optimize the

processor for particular areas. A large variance resulting from the different execution phases will defeat any attempts of optimizing a processor (or microcode used for implementation) even for a single program.

The statistical model presented in the next section is designed to address precisely these questions. As a by-product, it will yield the instruction mix composed of over 10 million instructions selected at random from a large number of PDP11 programs. We now describe an experiment designed to study the variation in the instruction mix caused by the three sources. This type of design is well known in statistics as "Nested (hierarchical) design" (see [ANDE74], [SNED67] chapter 10). Consider the instruction mix as a vector of the fractional utilizations of different opcodes arranged in some fixed order. The model being used is as follows: [4]

$$opcode^{i}_{jkl} = \mu^{i} + A^{i}_{j} + P^{i}_{(j)k} + S^{i}_{(jk)l}$$

Where,

i = 1,2, ... N (N = number of different opcodes)
j = 1,2, ... a (a = number of different areas)
k = 1,2, ... p (p = number of different programs within an area)
l = 1,2, ... s (s = number of different segments within a program within an area)

and

μ^{i} = overall mean fraction for the i^{th} opcode

A^{i}_{j} = effect of the j^{th} area on the i^{th} opcode

---

[4] Actually, our experiment can also be considered as an example of a 'Mixed nested (hierarchical) model'. The reason is that our 5 application areas have not been chosen at random from a large number of available application areas. They are really fixed areas that we want to investigate. The programs within each area and the segments within each program are however random samples from a large set. The analysis does not change under the 'fixed' model, only the interpretation of the results changes. Whereas in the 'random' model we can extend our results to all application areas on a PDP-11, in the 'fixed' model, the conclusions drawn are restricted only to these specific areas. However, since the concept of an application area is vague and since we have covered almost all application areas on the PDP-11 with our 5 areas, we have decided to ignore the distinction between a 'random' and a 'fixed' factor model.

$P^i_{(j)k}$ = effect of the k^{th} program in the j^{th} area on the i^{th} opcode

$S^i_{(jk)l}$ = effect of the l^{th} segment in the k^{th} program in the j^{th} area on the i^{th} opcode

The quantities A, P and S have the following distributions:

A^i_j is taken from $\mathbb{N}(0,\sigma_A)$ for all i

$P^i_{(j)k}$ is taken from $\mathbb{N}(0,\sigma_P)$ for all i,j

$S^i_{(jk)}$ is taken from $\mathbb{N}(0,\sigma_S)$ for all i,j,k

where, $\mathbb{N}(\alpha,\beta)$ represents a normal distribution with mean α and variance β.

The analysis of variance table (ANOVA) for each opcode is as follows:

Table 4.4 The ANOVA Table

Source of variance	Degrees of freedom	Expected Mean square
Application areas	a-1	$\sigma_S^2 + s*\sigma_p^2 + s*p*\sigma_A^2$
Programs within an area	a*(p-1)	$\sigma_S^2 + s*\sigma_p^2$
Segments within programs	a*p*(s-1)	σ_S^2

Where,

σ_S^2 = Variance due to segments

σ_p^2 = Variance due to programs

σ_A^2 = Variance due to application areas.

Once the three variances are determined for an opcode, it is possible to compare them against each other to determine which of those are significant. The variances are estimated using the measured instruction mix data as described below.
Let us define -

$opcode^{i}_{jk*}$ = average fraction of the i^{th} opcode over all segments for the k^{th} program in the j^{th} area.

$opcode^{i}_{j**}$ = average fraction of the i^{th} opcode over all programs for the j^{th} area.

$opcode^{i}_{***}$ = average fraction of the i^{th} opcode over all areas.

The formulas for the sums of squares are as follows:

Table 4.5 The Measured Sums of Squares

Sum of squares between--	formula for sum of squares
application areas	$s*p* \sum_{1 \le j \le a} (opcode^{i}_{j**} - opcode^{i}_{***})^2$
programs within areas	$s* \sum_{1 \le j \le a} \sum_{1 \le k \le p} (opcode^{i}_{jk*} - opcode^{i}_{j**})^2$
segments in programs	$\sum_{1 \le j \le a} \sum_{1 \le k \le p} \sum_{1 \le l \le s} (opcode^{i}_{jkl} - opcode^{i}_{jk*})^2$

The sums of squares are divided by the corresponding degrees of freedom giving the mean squares. The ANOVA table (table 4.4) shows that the mean squares for segments within programs directly estimates σ^2_S. The other two mean squares can be used to estimate σ^2_P and then σ^2_A.

4.4. Testing the Null Hypothesis

There are two null hypotheses for every opcode that can be tested with our design of the experiment.

Hypothesis 1: There is no difference in the fraction of use of the opcode from application area to application area, that is,

$$\sigma_A^2 = 0$$

Hypothesis 2: There is no difference in the fraction of use of the opcode from program to program within a given application area, that is,

$$\sigma_P^2 = 0$$

These hypotheses can be tested using the analysis of variance procedure. Consider the ratio (see table 4.4)-

$$F1 = \frac{\text{Mean square between areas}}{\text{Mean square between programs}} = \frac{\sigma_S^2 + s*\sigma_P^2 + s*p*\sigma_A^2}{\sigma_S^2 + s*\sigma_P^2}$$

If hypothesis 1 holds ($\sigma_A^2 = 0$), the ratio F1 should be 1. However, if the hypothesis is false ($\sigma_A^2 > 0$) , F1 should be > 1. Moreover, greater the dependence of the fractional use of the opcode on the application area, larger will be the value of σ_A^2 and

larger will be the the value of F1 [5]. Fisher has tabulated the theoretical 1% points for the F ratio distribution for various degrees of freedom in the numerator and denominator. The observed F value exceeds this tabulated value in 1% of the cases even when the null hypothesis is true. The tabulated value [6] is used in the following way : If the observed F is much less than the tabulated value, then the null hypothesis holds. On the other hand, if the observed value is larger than the tabulated value, then we can say with 99% confidence that the null hypothesis is false (in other words, the variance is significant at the 1% level). When the observed value is only slightly less than the tabulated value , we cannot make a strong statement regarding the null hypothesis.

To test hypothesis 2, we use the variance ratio -

$$F2 = \frac{\text{Mean square between programs}}{\text{Mean square between segments}} = \frac{\sigma_S^2 + s\sigma_P^2}{\sigma_S^2}$$

Whenever the above method suggests that a particular variance is significant at the 1% level, it is interesting to estimate the confidence limits around the *measured* value of the variance (the formulas for the upper and lower bounds are given on page 285 in [SNEDO67]). This tells us how much variability can be expected in the *measured* variance if we were to do the whole experiment again. If we perform the experiment with more application areas or with more programs in each area, these confidence

---

[5] the variance ratio is usually denoted by F. We will call the above ratio F1 and the corresponding ratio for hypothesis 2, F2

[6] In our experiment, $n=5$ and $p=5$. So the 1% point for F1 is 4.43 and the 1% point for F2 is 1.91

limits will shrink but the variance caused by these factors in an opcode will not be too different. Since the instruction mix for a segment is not composed of smaller measurements, we cannot determine the confidence limits around the variance due to the segments.

4.5. Details of the Experiment

In the actual design of the experiment we faced the following trade off: to gather an instruction mix representative of the instruction execution of the whole PDP11 family of computers, we need to measure a large number of application areas from all the areas in which a PDP11 has been used; but on the other hand, as the number of application areas goes up, the time required to gather and run a few (at least 5) representative programs in each area also goes up. We have therefore restricted our experiment to five areas which represent most of the instruction executions on the PDP11 family of computers. Within each area, we have selected five representative programs that were being used by other users, that is, we did not study synthetic benchmarks. Within each program, we measured 24 segments and each measurement consisted of sampling 20000 instructions at random and constructing the instruction mix vector. The following areas and programs were used :

1. Scientific Fortran benchmarks: 5 user benchmarks
2. Business Cobol benchmarks: 4 user benchmarks, plus 1 sort
3. Operating systems: RSX11-M, RSX11-D, RT11, RSTS, Hydra
4. Systems programs: Fortran and Cobol compilers, BASIC interpreter, macro assembler and the linker.
5. Device oriented systems: graphics terminal, front end processor, Xerographic

printer controller, processor 0 on C.mmp (heavily loaded with I/O devices), CM* host (controlling a large collection of LSI-11 processors).

Since the operating systems and the real time device oriented systems were to be studied, it only tool that was applicable for all the areas was a hardware monitor. This let us measure the instruction mix without perturbing the measured system in any way. Because of limitations of the hardware monitor used (K.mon), we could not record every instruction taking place on the measured processor. Such a trace of instructions requires a very high bandwidth output device for receiving the data from the hardware monitor. We were therefore restricted to sampling instructions at random. There is actually no simple way to specify selection of instructions at random in K.mon. K.mon can be set up to select every n^{th} instruction occuring on the P.host unibus, where $1 \le n \le 2^{16} - 1$. Unfortunately the value of n has to be specified before the experiment begins and it then remains constant throughout the experiment. We usually chose a prime number for n (typically 31 or 127) so that the problem of K.mon synchronizing with small loops on the P.host was avoided. There is still the possibility of a loop of some exact multiple of n instructions, but we felt that this problem is not significant since in our experiments such loops were broken up due to the following events:

a> occurrence of a clock or a device interrupt

b> a pause in the data gathering due to overflowing K.mon's internal data gathering buffers. The overflows occur because of the slow output link.

It is also possible to sample the instructions such that the first instruction occuring after every n microseconds is selected for analysis. This method is however, not

suitable for measuring the instruction mix since it actually gives the distribution of the time spent in executing the various instructions instead of the frequency of their execution. Since we could not obtain a record of consecutive instructions, a study of frequently executed instruction sequences could not be performed. In chapter 6 we describe a separate experiment to study the instruction sequences.

Even when sampling instructions at random, it was not possible to record the occurrence of every sampled instruction because of the low speed of our output device (a 300 baud link to the PDP-10). We had to perform some data compression in the supervisory computer before storing the data for post-processing. Each sampled instruction is used to update the appropriate counters in the main memory of the supervisory computer. After 20000 instructions are processed (i.e. one segment), the counter values are stored on some output device for later processing. The counters are maintained for the following: each of the PDP11 opcodes, 8 modes and 8 registers for single operand instructions, 8 modes and 8 registers each for the source and the destination operand for double operand [7] instructions. This data compression reduces the amount of data that has to be transmitted to the output device for post-processing, but it prevents us from studying the occurrence of cross-products of addressing modes and registers or of source and destination modes. This study therefore cannot answer questions like how many times either the source or the destination mode is zero for the MOV instruction?

Most of our measurements were performed on the PDP 11 models 20 and 40. These

---

[7] The single operand instructions are CLR(B), COM(B), INC(B), DEC(B), NEG(B), TST(B), ROR(B), ROL(B), ASR(B), ASL(B), SWAB, ADC(B), SBC(B), SXT The double operand instructions are MOV(B), CMP(B), ADD,SUB, BIT(B), BIC(B), BIS(B)

models do not have a sophisticated floating point or business instruction set. If we had used the newer models, we would have certainly discovered more usage of the floating point and business specific instructions in the scientific and business application areas. Moreover, the Fortran and Cobol compilers have been improved since the measurements were performed and the new compilers are expected to use the instruction set more wisely.

4.6. Results of the Instruction Mix Experiment

The results of our experiment are presented in Table 4.6. Only those opcodes and addressing modes which show significant use (more than 0.01 percent) are included in the table. The complete instruction mix is given in appendix B. For the variance due to application areas and programs, the 90% confidence limits around the variance are given in square brackets. If the variance is larger than 100, the confidence limits are omitted. The variance is reported as 0 if it is less than 0.001. Some values of the variance are negative and these values and their confidence limits are not given. A negative variance means that the variation is less than what would be expected if the opcode fractional usage values were drawn from a single normal distribution. We can interpret the negative variance to mean that the variance is small. The total variance for an opcode is always positive. Our particular model attempts to split this total variance into three parts. It just so happens that the particular values of data collected sometimes lead to a negative value for one of these three parts. The observed F1 and F2 values are also given for each opcode and the addressing modes and registers. The significant F values are flagged with a '*'. Note that all the variances due to programs are significant but quite a few variances due to application areas are not. It is also

interesting to observe how the instructions and the addressing modes are being used by programs in each application area. Table 4.7 presents the instruction mix by application area.

Table 4.6 The Instruction Mix and its Variance

Number of application areas: 5
Number of programs per appl area: 5
Number of segments per program: 24
Number of instructions per segment: 20000
Total number of instructions sampled: 12 million

Name	Overall mean	Overall standard deviation	variance due to application area	variance due to programs	variance due to segments	F1	F2
MOV	31.205	2.170	8[0,22]	75[47,82]	8.258	1	* 220
BNE	5.597	1.675	1[2,19]	12[8,14]	4.236	* 5	* 73
BEQ	4.140	.931	3[1.0,5]	3[2,3]	2.776	* 5	* 31
CMP	4.005	.902	3[1,5]	2[1,2]	1.416	* 7	* 45
JMP	3.940	3.128	48[19,75]	3[2,3]	.420	* 67	* 206
DEC	3.919	1.577	10[2,17]	11[7,12]	1.416	* 5	* 198
ADD	3.684	.092		4[3,5]	.498	0	* 232
JSR	3.622	.818	2[.8,4]	2[1,3]	.555	* 5	* 123
TST	3.492	1.066	3[0,6]	10[6,11]	11.44]	2	* 22
ASL	3.413	.793	1[0,3]	8[5,9]	.445	1	* 472
BR	3.134	1.299	7[2,12]	3[2,3]	.393	* 12	* 203
RTS	3.008	.718	2[.7,3]	1[1.0,1]	.342	* 8	* 110
MOVB	2.024	.591	1[.3,2]	1[1,2]	1.052	* 4	* 43
CLR	1.675	.121		.9[.6,1]	.123	.4	* 181
BIT	1.675	.531	1[.2,1]	1[1.0,1]	.968	* 4	* 38
BCS	1.587	.500	.8[0,1]	2[1,2]	.479	2	* 104
INC	1.400	.230	.1[0,.4]	1[.9,1]	.136	1	* 242
SOB	1.331	1.041	4[1,7]	4[2,4]	.758	* 6	* 137
BIC	1.264	.361	.4[0,.8]	1[.9,1]	.163	2	* 193
BPL	1.233	.885	1[0,3]	12[7,13]	14.499	1	* 20
BGT	1.174	.222		1[1,2]	.828	.6	* 55
TSTB	1.169	.510		6[3,7]	10.540	1.0	* 15
SUB	1.105	.304	0[0,.4]	2[1,2]	.888	1	* 56
ROL	1.056	.207		2[1,2]	.962	.8	* 55
BMI	.886	.324	0[0,.4]	2[1,2]	1.161	1	* 44
BLE	.735	.100		.7[.5,.8]	.074	.3	* 236
BCC	.619	.284	.4[.1,.6]	.2[.2,.3]	.274	* 7	* 22
BITB	.574	.174	.1[0,.2]	.2[.1,.2]	.016	3	* 319
BHI	.563	.254	.3[0,.4]	.2[.1,.3]	.243	* 6	* 24
CLRB	.549	.207	.2[0,.3]	.2[.2,.3]	.073	4	* 74
CMPB	.504	.193	.1[0,.3]	.2[.1,.2]	.100	* 5	* 43
FADD	.478	.478	1[.5,1]	.1[0,.1]	.030	* 50	* 91

BGE	.463	.112	0[0,0]	.2[.1,.2]	.028	1	* 172
BIS	.460	.152	0[0,.1]	.2[.1,.2]	.236	3	* 19
FMUL	.420	.420	.9[.3,1]	0[0,0]	.024	* 284	* 15
SWAB	.414	.205	.2[0,.3]	.2[.1,.3]	.143	4	* 40
BLT	.409	.049		.1[0,.1]	.026	.6	* 96
BOR	.389	.112	0[0,0]	0[0,.1]	.044	3	* 51
RTI	.290	.105	0[0,0]	0[0,0]	.058	3	* 33
NOP	.281	.130	0[0,.1]	.1[0,.1]	.080	3	* 41
ASR	.235	.072	0[0,0]	0[0,0]	.008	3	* 28
BISB	.224	.096	0[0,0]	0[0,0]	.010	2	* 93
MFPI	.200	.131	0[0,0]	.2[.2,.3]	.013	1	* 454
MTPI	.175	.130	0[0,0]	.3[.2,.3]	.005	1	* 1364
COND	.151	.105	0[0,0]	0[0,0]	.032	* 4	* 46
BLOS	.148	.083	0[0,0]	0[0,0]	.035	2	* 57
ADC	.115	.050	0[0,0]	0[0,0]	.104	1	* 8
NEG	.102	.027		0[0,0]	.001	.8	* 375
FDIV	.099	.099	0[0,0]	0[0,0]	.001	3	* 1160
DECB	.097	.040	0[0,0]	0[0,0]	.004	2	* 110
INCB	.096	.030	0[0,0]	0[0,0]	.003	1	* 131
FSUB	.082	.082	0[0,0]	0[0,0]	.001	3	* 1734
TRAP	.073	.036		0[0,0]	.005	.7	* 248
ASLB	.066	.053	0[0,0]	0[0,0]	.021	1	* 79
BICB	.048	.011	0[0,0]	0[0,0]	.001	1	* 94
WAIT	.039	.021	0[0,0]	0[0,0]	.003	2	* 28
ASHC	.035	.026	0[0,0]	0[0,0]	.000	2	* 752
EMT	.033	.022	0[0,0]	0[0,0]	.000	* 8	* 116
RORB	.025	.011		0[0,0]	.001	.6	* 84
ASH	.023	.023	0[0,0]	0[0,0]	.000	2	* 435
DIV	.020	.008		0[0,0]	.000	.8	* 159
COM	.019	.013	0[0,0]	0[0,0]	.003	4	* 7
ASRB	.017	.016	0[0,0]	0[0,0]	.000	2	* 688
SBC	.017	.009	0[0,0]	0[0,0]	.001	2	* 28
SXT	.012	.007		0[0,0]	.000	.9	* 624
ADCB	.010	.006	0[0,0]	0[0,0]	.000	* 4	* 32
MUL	.010	.007	0[0,0]	0[0,0]	.000	3	* 126

Total percent of single operand instructions: 18.2 percent

Percentage of individual modes and register usage for single operand instructions:

mode 0	50.020	5.036	71[0,145]	276	60	2	* 111
mode 1	11.794	1.618	6[0,12]	32[19,35]	76	1	* 11
mode 2	8.498	1.205	2[0,6]	25[16,28]	6	1	* 95
mode 3	7.790	2.909		230	89	.9	* 62
mode 4	4.498	.909	.3[0,3]	23[14,25]	1	1	* 533
mode 5	.000	.000			0	0	0
mode 6	16.130	3.107	34[3,60]	68[42,75]	45	3	* 36
mode 7	1.281	.721	1[0,2]	6[4,7]	1	1	* 119
reg 0	20.191	3.987	39[0,86]	197	50	1	* 94
reg 1	11.368	1.998	5[0,17]	70[44,78]	58	1	* 30
reg 2	11.563	2.143	12[0,25]	53[34,59]	8	2	* 160
reg 3	16.078	8.547	329	175	31	* 10	* 136
reg 4	6.456	1.264	1[0,7]	29[18,32]	5	1	* 123
reg 5	5.437	1.824	10[0,19]	31[19,34]	9	2	* 77
reg 6	15.823	3.441	4[4,75]	85[54,94]	22	3	* 96
reg 7	13.014	3.655	19[0,60]	233	100	1	* 56

Total percent of double operand instructions: 46.8 percent

Percentage of source mode and register usage:

mode 0	24.784	4.869	103	42[26,46]	22.958	* 13	* 45
mode 1	8.211	2.287	24[8,38]	8[5,9]	6.810	* 14	* 31
mode 2	43.086	4.420	87[28,136]	49[30,54]	58.259	* 9	* 21
mode 3	6.510	2.505	24[5,41]	32[20,35]	18.197	* 4	* 43
mode 4	1.301	.431	.5[0,1]	2[1,2]	.552	2	* 95
mode 5	.000	.000		0[0,0]	0	.8	* 2
mode 6	15.536	1.460	7[1,13]	13[8,15]	9.993	3	* 34
mode 7	.572	.198	.1[0,.2]	.4[.2,.4]	.083	2	* 113
reg 0	15.736	.986		33[20,36]	25.454	.7	* 32
reg 1	15.605	2.985	25[0,51]	94[60,104]	11.938	2	* 191
reg 2	6.352	1.188	5[1,9]	6[4,7]	2.805	* 5	* 57
reg 3	6.537	1.505	9[2,14]	9[5,10]	19.779	* 5	* 12
reg 4	11.816	6.098	177	42[27,47]	2.383	* 21	* 432
reg 5	5.101	.684	.1[0,1]	10[6,11]	8.628	1	* 30
reg 6	15.866	1.994	9[0,21]	53[34,59]	7.556	1	* 171
reg 7	22.986	4.726	101	45[27,50]	75.249	* 11	* 15

Percentage of destination mode and register usage:

mode 0	42.361	4.018	73[25,113]	32[20,36]	46.330	* 11	* 17
mode 1	7.211	.540		9[5,10]	8.535	.7	* 27
mode 2	15.478	6.163	172	87[55,96]	28.686	* 10	* 74
mode 3	5.003	1.834	7[0,16]	45[28,50]	51.738	1	* 22
mode 4	18.192	5.607	152	22[13,24]	6.980	* 35	* 76
mode 5	.017	.012	0[0,0]	0[0,0]	.002	1	* 39
mode 6	11.017	2.723	31[3,52]	27[17,29]	3.738	* 6	* 175
mode 7	.701	.168	0[0,.1]	.5[.3,.5]	.220	1	* 54

reg 0	16.978	3.004	36[9,60]	41[26,45]	25.753	* 5	* 39
reg 1	14.918	1.965	10[8,21]	44[28,48]	6.447	2	* 165
reg 2	13.440	5.113	123	36[23,48]	13.294	* 17	* 67
reg 3	7.914	1.410	7[1,13]	11[7,12]	5.883	4	* 47
reg 4	6.866	1.535	9[2,15]	12[7,13]	5.520	* 4	* 55
reg 5	5.861	1.431	6[.4,12]	16[10,18]	1.824	3	* 216
reg 6	22.197	5.894	161	60[38,66]	10.863	* 14	* 134
reg 7	11.827	3.806	58[13,95]	68[42,75]	72.424	* 5	* 23

Table 4.7 Instruction Mix by Application Area

opcode	area 1 Scientific	area 2 Business	area 3 Operating Systems	area 4 Systems Programs	area 5 Real time Systems
MOV	36.859	24.784	30.079	29.161	35.143
BNE	.898	6.916	10.772	6.098	3.301
BLO	1.428	5.085	2.612	6.601	4.974
CMP	1.666	6.105	2.647	6.085	3.522
JMP	16.430	.484	1.159	1.277	.350
DEC	2.692	4.141	9.897	1.255	1.611
ADD	3.791	3.709	3.546	3.424	3.952
JSR	1.020	5.183	2.984	3.372	5.549
TST	1.790	2.774	1.468	4.063	7.356
ASL	6.373	3.586	1.782	2.591	2.724
BR	8.293	2.048	1.521	1.466	2.319
RTS	.413	4.094	2.992	2.989	4.553
MOVB	.385	3.159	3.364	2.263	.950
CLR	1.264	1.996	1.818	1.626	1.673
BIT	.037	1.696	1.582	3.393	1.667
BCS	1.679	2.723	.650	2.618	.264
INC	1.930	1.940	.378	1.587	1.210
SOB	.007	.251	5.456	.836	.106
BIC	.058	1.053	2.083	1.199	1.926
BPL	.020	.362	.288	.756	4.740
BGT	.655	1.515	1.847	1.040	.812
TSTB	.095	.460	.974	1.275	3.040
SUB	.394	.822	2.191	1.258	.861
ROL	1.948	1.359	.686	.814	.471
BMI	.117	.364	1.535	1.756	.655
BLE	.899	.414	.783	.613	.968
BCC	.222	1.674	.221	.762	.216
BITB	.008	.567	1.084	.506	.707
BHI	.061	1.140	.053	1.207	.360
CLRB	.558	1.257	.185	.654	.092
CMPB	.026	.974	.563	.858	.097
FADD	2.390	.000	.000	.000	.000
BGE	.290	.234	.846	.370	.576
BIS	.192	.328	.132	.828	.818
FMUL	2.000	.000	.000	.000	.000
SWAB	.323	1.205	.033	.314	.195
BLT	.355	.512	.542	.338	.298
BOR	.483	.544	.003	.442	.074

BTJ	.047	.247	.091	.519	.546
NOP	.007	.746	.250	.333	.072
ASR	.073	.487	.171	.289	.157
BTSB	.101	.465	.069	.193	.229
MFPI	.000	.355	.000	.645	.000
MTPI	.000	.201	.000	.672	.000
COND	.007	.543	.001	.206	.000
BLOS	.011	.463	.072	.360	.032
ADC	.007	.269	.173	.107	.012
NEG	.177	.034	.123	.048	.125
FDIV	.434	.000	.000	.000	.000
DECB	.011	.176	.000	.194	.106
INCB	.007	.034	.000	.188	.111
FSUB	.408	.000	.000	.000	.000
TRAP	.000	.052	.093	.015	.203
ASLB	.005	.047	.003	.276	.000
RTCB	.005	.050	.059	.073	.054
WAIT	.016	.063	.000	.006	.111
ASHC	.134	.000	.000	.039	.000
LMT	.002	.041	.117	.007	.000
RORB	.004	.023	.000	.047	.052
ASH	.001	.000	.001	.115	.000
DIV	.033	.001	.000	.038	.030
COM	.000	.006	.000	.028	.000
ASRB	.003	.001	.000	.002	.000
SBC	.002	.041	.004	.039	.000
SXT	.029	.000	.000	.000	.030
ADCB	.004	.032	.000	.015	.000
MUL	.035	.000	.000	.015	.000
ROLB	.000	.045	.000	.000	.000
RTT	.000	.000	.000	.013	.000
BVS	.000	.001	.000	.003	.000
COMB	.000	.002	.000	.002	.000
IOT	.000	.000	.000	.003	.000
NEGB	.000	.000	.000	.001	.000
XOR	.000	.000	.000	.001	.000
BVC	.000	.000	.000	.000	.000

Addressing mode usage in single operand instructions:					
Single operand	17.624	19.930	17.682	16.359	18.844
Sing mode 0	54.781	57.845	61.533	57.477	38.805
Sing mode 1	15.336	7.483	8.551	12.363	14.745
Sing mode 2	7.972	9.526	6.972	17.534	5.465
Sing mode 3	6.184	2.182	7.200	4.413	18.382
Sing mode 4	4.996	2.631	4.710	3.342	2.389
Sing mode 5	.000	.000	.000	.000	.000
Sing mode 6	7.350	20.348	10.967	24.484	17.450
Sing mode 7	3.382	.064	.068	.287	2.065
Sing reg 0	24.767	20.585	5.318	21.495	28.791
Sing reg 1	13.456	14.436	4.448	9.303	15.137
Sing reg 2	11.516	18.911	6.583	12.679	8.111
Sing reg 3	7.035	5.183	58.889	10.332	7.821
Sing reg 4	10.809	5.642	5.671	6.987	5.822
Sing reg 5	.748	7.581	4.473	11.326	3.357
Sing reg 6	29.081	12.587	15.870	11.426	10.281
Sing reg 7	2.585	15.234	7.324	10.457	25.480
Addressing mode usage in double operand instructions:					
Double operand	43.583	43.711	47.398	44.348	43.926
src mode 0	10.216	35.980	16.524	31.166	20.032
src mode 1	17.038	7.669	3.913	6.216	6.228
src mode 2	37.760	38.723	60.733	38.774	33.437
src mode 3	14.365	.738	6.672	2.065	8.111
src mode 4	1.972	1.363	.515	2.477	.176
src mode 5	.000	.001	.000	.000	.000
src mode 6	17.942	14.444	10.583	18.800	15.853
src mode 7	.167	1.021	1.053	.441	.171
src reg 0	16.706	15.326	18.205	16.181	12.264
src reg 1	14.843	14.198	27.078	10.262	11.649
src reg 2	2.680	9.345	4.647	7.624	7.464
src reg 3	1.276	8.827	3.555	5.239	7.787
src reg 4	35.963	6.349	4.020	8.642	4.102
src reg 5	2.901	6.587	4.794	6.529	4.691
src reg 6	20.072	14.192	9.970	14.501	20.597
src reg 7	5.547	25.177	21.740	31.021	31.446
dest mode 0	33.530	50.206	28.812	42.562	50.672
dest mode 1	8.283	7.844	5.689	8.030	6.135
dest mode 2	6.143	12.419	38.512	12.639	6.068
dest mode 3	3.335	1.785	3.863	3.448	12.186
dest mode 4	48.102	12.151	8.565	15.025	15.113
dest mode 5	.000	.005	.000	.004	.016
dest mode 6	1.726	14.513	13.353	17.125	8.366

dest mode 7	.208	1.815	.405	1.[illegible]	.[illegible]
dest reg 0	26.572	17.709	8.[illegible]	17.[illegible]	18.078
dest reg 1	9.801	15.[illegible]	21.[illegible]	17.[illegible]	14.775
dest reg 2	2.142	14.150	32.[illegible]	18.137	8.398
dest reg 3	3.125	10.285	11.187	7.735	7.239
dest reg 4	10.100	6.889	2.371	10.[illegible]	4.091
dest reg 5	2.858	9.493	5.098	8.809	3.847
dest reg 6	44.977	14.517	12.079	18.713	20.699
dest reg 7	1.074	11.603	6.129	19.034	21.293

4.7. Conclusions

In this chapter we discussed a major experiment designed to address the question of the variability of the instruction mix. The measurements indicate that a statistically significant variation in the instruction mix is caused both by the application area and the individual programs in a given area, but not by the different phases of execution in the same program. Moreover, different instructions exhibit different behavior. The most heavily used instruction (MOV) is affected more by the individual programs than the application areas. In other words, we cannot speed up the execution of the MOV instruction and hope to achieve the same level of speedup for all programs even in a specific application area. Similar conclusions can be drawn for other instructions as well. It is therefore not proper to attempt to over-optimize a processor for a particular application area.

The overall means and the standard deviations reported for individual instructions are important in their own right. The overall standard deviation and the individual variances are to be used as follows:

> Suppose we make another measurement of the instruction mix using A1 areas, P1 programs in each of the areas and S1 segments in each program. Note that any of the quantities A1, P1 or S1 can be equal to 1. We then calculate the percentage of usage of say the MOV instruction to be M1. We can then say that M1 as an estimate of the overall percentage of usage of the MOV instruction for all the executions on a PDP-11 has a variance equal to -

$$\sigma_A^2/A1 + \sigma_P^2/(A1*P1) + \sigma_S^2/(A1*P1*S1)$$

Where, the variances are those given in table 4.6 for the MOV instruction.

Figure 4.1 displays the PDP-11 instruction frequency distribution measured in our study. It is interesting to observe that only 10 instructions account for about 70 percent of the instructions executed on a PDP-11. It can be seen from Table 4.6 that many of the instructions are seldom used. Similar results have been reported by other researchers studying the instruction mix. Our measurements indicate that the addressing mode 5 (auto-decrement deferred) is almost never used and a good case can be made for its elimination. Since our study involves sampling more than 10 million instructions from 25 independent programs, we expect that the true nature of the instruction mix for the PDP-11 has been captured. Our results form a data base which has important applications in the design, implementation and emulation of PDP-11 and similar processors.

The measurements reveal some anomalies. It can be seen that the σ_A^2 for the JMP instruction is extremely high. A look at the raw data displayed in the instruction mix by application area table (table 4.7) indicates that the application area consisting of scientific Fortran benchmarks shows very high use of the JMP instruction compared to any other application area. This leads to the large value of the observed variance caused by application areas for this instruction. The trouble lies in the particular

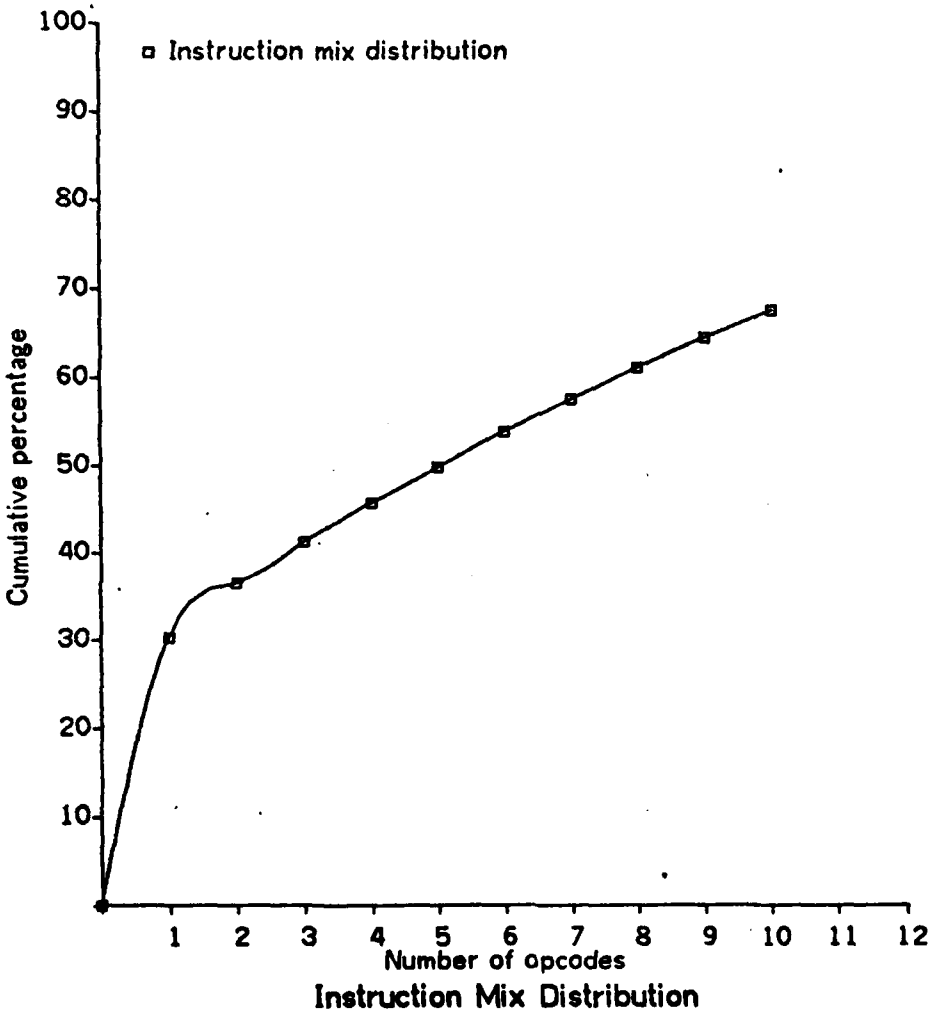


Figure 4.1 Distribution of Instruction Usage

Fortran compiler[8] used which generates the JMP instruction instead of the branch instructions.

It is interesting to look at the whole experiment in the light of the workload characterization problem. Intuitively the different application areas represent different workloads since it is clear that each of these areas is doing a different kind of 'work'. The Fortran programs are manipulating numbers for the purpose of solving equations, the operating systems are performing the processor and memory scheduling functions whereas the real time systems are responding to the events happenning in their environments. Our experiment is an attempt to characterize these intuitively different workloads in terms of their instruction mixes. It turns out that a meaningful characterization at such a low level is not possible due to the variation in the programs belonging to these areas. This negative result should not be interpreted as saying that a characterization at a higher level is not possible; in fact, future research should concentrate on the next higher level of atomic 'work' e.g. in terms of manipulations of higher level data structures like vectors, lists, process control blocks, strings and so on instead of integers, reals, bits and words as was done in this study.

---

[8] We used the FORTRAN-IV compiler which has now been replaced by FORTRAN-IV+.

5. Multiprocessor Contention for Shared Data

5.1. Introduction

This chapter studies contention for shared data objects in a multi-processor system. A common problem in a multi-processor system is the contention among processors for shared resources. This contention can occur at various levels. C.mmp has up to 16 PDP-11's which can independently access 16 memory ports through a cross-point switch. The lowest level contention therefore occurs at the cross-point switch. If two or more processors try to access the same memory port at the same time, all but one of them have to wait. This problem has been studied earlier by Strecker[STRE70], Bhandarkar [BHAN73] , McCredie[MCCR73] and by Basket and Smith [BASK76]. Fuller [FULL76] applied these models to specific hardware configurations of C.mmp and showed that memory interference does not cause severe degradation, i.e. less than 10 percent. On a higher level, there is contention for shared data. The shared data can be a few bytes in a system table or a large data structure like a linked list or a file. On a still higher level, there is contention for devices like the line printer or disks and common software processes for memory management or user message handling. In this chapter we investigate the contention for the shared data structures in Hydra.

The problem of contention for shared data is somewhat more difficult for a multiprocessor than for a uniprocessor. In a uniprocessor, system integrity can be maintained by simple techniques like blocking all interrupts while accessing critical system tables and by careful coding of the interrupt routines. In a multiprocessor however, the scheduling and coordination of the individual processors to achieve parallel operation is a significant problem. One approach is to have a common shared

database which contains all the information necessary for a processor to make its scheduling decisions. This is the approach taken by Hydra- the kernel of the operating system for C.mmp. While one processor is examining or updating this database, all others must be prohibited from accessing or modifying it. There are other shared data objects as well which are also required to be accessed by only one processor at a time. The mechanism for such mutual exclusion in C.mmp/Hydra system is a 'lock'.

A Hydra lock should not be confused with a semaphore since the former is at a more primitive level than the latter. In Hydra, the locks are used to synchronize access to small but often frequently accessed shared data objects. The 'lock' and 'unlock' operations are similar to the P and V operations on semaphores except that when a processor blocks while trying to set a lock, the process running on the processor is not context-swapped off the processor. Rather, the processor is simply put in the wait state (the processor is said to be blocked) until the receipt of an inter-processor signal (interrupt) notifying it that the lock has been reset.

The lock and unlock operations are used to implement a more general and sophisticated synchronization mechanism and some message systems. The fundamental question we sought to investigate in this study was how much processor degradation is due to contention for shared data objects in Hydra synchronized by the lock/ unlock operations. The amount of degradation will be affected by the number of active processors, lengths of critical sections (i.e. the instructions executed between a lock/unlock pair), frequency of lock execution and the distribution of lock/unlock operations across the different shared data objects.

5.2. Review of Previous Work

The software lockout problem was modelled as early as in 1968 by Madnick [MADN68]. He considered a simple system consisting of a single critical section and calculated the mean number of blocked processors as a function of the total number of processors in the system. McCredie [MCCR73] extended this model to include an arbitrary number of critical sections in tandem. The results showed that it is advantageous to have two smaller critical sections instead of a single critical section which does the work of the two smaller ones. The designers of Hydra have therefore chosen to have many small critical sections in Hydra.

To the best of our knowledge, experimental verification of any of these models has not been attempted so far. The software lockout problem cannot be investigated using software monitors due to the excessive perturbation involved. The fact that a powerful hardware monitor is necessary for this study is probably the reason why this important problem could not be examined earlier.

5.3. Description of the Experimental Setup

The Kmon was used in the tracing mode to record the occurrences of all events related to locks. The data was post-processed to reconstruct the operations on the various locks and the blocking experienced by the host processor.

A lock is composed of three fields: lock count, sublock and mask. The lock count is initially 1 indicating that the lock is free. When the shared object becomes locked, the lock count is -N, where N ($N \geq 0$) is the number of processors waiting for the object. The sublocks are used to ensure that only one of the waiting processors get access to

the shared object when it becomes free. The mask field is used to indicate which processors are blocked on the lock. The schematic code sequences for lock and unlock are given below:

```
LOCK:     decrement lock count.        UNLOCK: increment lock count.
          exit if equal to 0.                  exit if greater than 0.
BLOCK:    mark self as blocked.                initialize sublock.
          Turn off all interrupts              Send unblock interrupt to all
            except the unblock                   processors blocked on this lock
            signal.                            Exit.
          wait.
          try for sublock.
          If fails go to BLOCK.
UNBLOCK:  exit.
```

The monitor was set up to detect events as follows:

(1) When a 'lock' is attempted, obtain address of the lock and time stamp.

(2) When an 'unlock' takes place, obtain address of the lock and time stamp.

(3) When an 'unblock' takes place, obtain the time stamp.

The first two events give the critical section time when the attempt for the lock is successful. Whenever event 3 happens, it is always after event 1. It indicates that the attempt for lock was unsuccessful and that the processor was 'blocked'. The address of the lock is obtained from the previous event 1. The blocked time is determined from the time stamps of event 1 and event 3.

We used three benchmarks to generate loads on the system. Each experiment consisted of running one benchmark by itself and collecting the output of the hardware monitor for post-processing. All the benchmarks create about 16 different processes, each executing the same program. The processes do different amounts of computation, synchronize with each other and repeat. Benchmark 1 and 2 are two versions of a parallel program to find the roots of a transcendental equation [8]. They use two different types of semaphores for synchronization among themselves. The third

benchmark is a synthetic program which executes various kernel calls intermixed with small amounts of user level computing. A fourth measurement was made during the usual user hours to give the frequency of usage of various locks for the current typical user load. At the present time, C.mmp is not heavily loaded during general user sessions and measurements of C.mmp under near saturation conditions will have to wait until general usage of C.mmp increases.

5.4. Locks in C.mmp/Hydra environment

The kernel of the operating system for C.mmp is known as Hydra. It has been described extensively in [WULF75]. We briefly summarize here the pertinent information. Hydra solves the processor scheduling and coordination problem by maintaining global data structures containing information regarding the status of processors and feasible processes. Locks have to be associated with the objects in this database. Apart from these locks, there are locks on other shared objects. Examples of an object are a page, a semaphore, a process or a file. Every object has one or more locks associated with it for accessing different parts of the object. Since there are thousands of objects in Hydra, there are also thousands of locks.

Not all locks are however, heavily used. In our experiments, we observed of the order of 5 frequently used locks and a number of lightly used locks. One of the most frequently used locks is a 'feasible queue' lock. Processes which are ready to run are placed in one of eight feasible queues waiting for a processor to become free. Currently only two of the eight queues are being used, the first one for regular processes and the eighth one for high priority processes. Another frequently used lock is on the 'processor list' which is a list of all 16 processors containing information

about their status. Every time a processor becomes free, it goes through the feasible queues and the processor list to determine which processor should work on the next process. The next important lock is on the free core list. It is the lock on a list of free physical page addresses. This list is used when pages are to be swapped out or brought in. A similar lock for storage inside the kernel is called the 'kernel storage lock'. The lock on 'stop mailbox' is used by the policy module [9] to communicate with the kernel. This lock is accessed every time a process has to be started or stopped. The 'KMPS lock' is the lock on a free core list which is used by the scheduler to allocate and deallocate fixed size blocks for process information. It is used every time a process is started or stopped or when a message is sent to a process.

There is another interesting lock called 'lock on a page'. Every data page (the size of a page is 8K bytes in C.mmp) in Hydra has a lock associated with the whole page and this lock is always at a fixed offset from the beginning of a page. Since there are so many pages in the system, they have to be overlayed through a relocation register. Due to the well-known deficiencies of a hardware monitor using information internal to an operating system (or other software systems), it is not possible to pinpoint the particular page that has been locked even when the lock is detected at the proper offset from the beginning of the page. All one can say is that some data page has been locked. The result is that even though locks under the common heading of 'lock on a page' are accessed a large number of times, the number of times a processor has to wait on such a lock is much less than if there were only one 'lock on a page' for all pages. This phenomenon has to be treated in a special way when developing a model.

---

[9] A policy module makes the decisions regarding resource allocation among user programs.

In addition to the above, there are a number of locks which are used very infrequently. Most of them occur in the overlay data pages and so the hardware monitor is not able to precisely determine which locks are being used. We grouped all these locks under the heading 'Miscellaneous'. These locks also have to be modelled in a special way.

The processors on C.mmp are non-homogeneous. Some are PDP11/40's and some are PDP11/20's. Also, some have I/O devices and some do not. However, since the Hardware monitor can monitor only one processor at a time, we were constrained to measure only one processor. There is a software tracer available on Hydra, which can be used to monitor all processors at once. It is, however, not suitable for studying critical sections since recording an event with the tracer takes about as much time as a typical critical section. The perturbation introduced by tracing is unacceptable for this study. In our experiments, we had more processes than processors, so all the processors were busy most of the time. We expect all the processors to exhibit behavior like the measured processor as far as critical sections are concerned. Table 5.1 presents the measurements for the three programs.

Table 5.1 Measurement of the Locking Behavior in Hydra

	Program 1	Program 2	Synthetic load of Program 3	General multiuser session
1. Average kernel instructions[10] between two successive locks	413	224	515	
2. Number of different locks detected	53	79	181	
3. Freq usage of specific locks:				
Processor list lock	0.1584	0.3007	0.1151	0.3420
feasible queue 1	0.1184	0.2829	0.1050	0.05995
feasible queue 8	0.0338	0.0056	0.0	0.0028
Lock on a page	0.1723	0.0	0.3943	0.234
Core lock	0.0457	0.0	0.0544	0.0614
Stop mail box	0.0826	0.0	0.004	0.022
KMPS lock	0.0927	0.0	0.005	0.0
Miscellaneous	0.2961	0.4108	0.2523	0.2312
4. Average time inside a critical section (microsec):				
Processor list lock	348	409	378.5	
feasible queue 1	191.5	239	259.5	
feasible queue 8	156	168.5	---	
Lock on a page	338	---	430.5	
Core lock	557.5	307.4	684.5	
Stop mail box	282	264.4	297	
KMPS lock	108.6	123	134	
Miscellaneous	317.5	461	441	
Average for all locks	279	378	279	
Run dependent data:				
5. number of active processors	13	14	12	
6. Total time of measurement(millisec)	17393	32924	20255	
7. Total # of times locked	2955	5041	4360	
8. Total # of times blocked	130	577	146	
9. % of locks that blocked	5.5%	11.7%	6.1%	
10. % time spent in kernel	61.8%	16.9%	37.7%	
11. Average time(microsec) between locks	5888	6531	4646	
12. % time spent in the blocked state	0.29%	0.83%	0.74%	

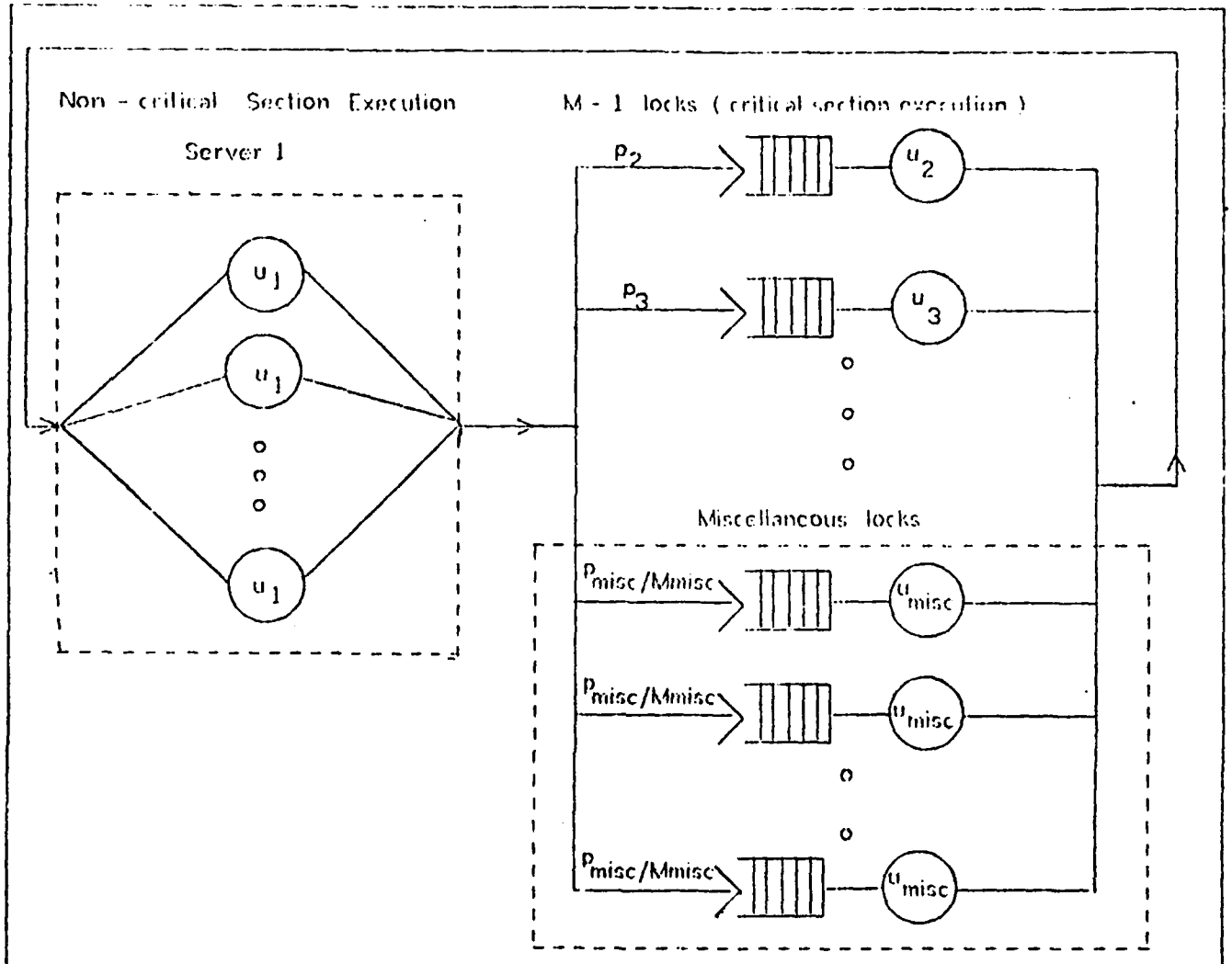
[10] For the processor under measurement, the average instruction execution time was 2.8 microseconds.

5.5. The model

Earlier studies of the software lockout problem had focussed on using a model with critical sections occuring in tandem. Our measurements on Hydra indicated that the locking behavior in Hydra approximates a model with critical sections in parallel instead of in tandem. Figure 5.1 displays the transition matrix for lock accesses observed for one program on Hydra. It lists the number of times a lock was called within 100 microseconds of exiting the previous lock. It can be seen that very few of the transitions occur in tandem.

Figure 5.1 Transition Matrix for Lock Accesses

Lock name	Total # times used	Transition to 1	2	3	4	5	6	7	8	9	10
1. Feasibility queue 1	350	0	0	0	0	0	0	0	0	0	191
2. Processor list	468	124	0	0	0	33	0	0	0	0	0
3. Kstr lock	70	0	0	0	0	0	0	0	66	0	0
4. Fdsk lock	0	0	0	0	0	0	0	0	0	0	0
5. Feasibility queue 8	100	0	0	0	0	0	0	0	0	0	16
6. Stop mail box	244	0	0	0	0	0	0	0	0	0	0
7. Core lock	135	1	0	0	0	0	0	2	0	0	0
8. Lock on a page	509	1	0	0	0	0	2	0	0	0	0
9. I/O system lock	66	0	0	0	0	0	0	0	0	0	0
10. Miscellaneous locks	109	0	1	1	0	0	0	0	0	0	6



u_1 = Average between locks on a single processor

u_i = Average time in the i^{th} critical section $\{ 2 \le i \le M \}$

Mmisc = Numbers of miscellaneous locks being modelled

p_i = Probality of attempting the i^{th} lock $\{ 2 \le i \le M - M_{misc} \}$

$p_{misc} = 1 - (p_2 + \ldots\ldots + p_{M - M_{misc}})$

Figure 5.2. The Central Server Model

A simple central server model ([BUZE73]) was therefore used to model the contention (see figure 5.2). In our model, the N customers are the processors in C.mmp, the service site 1 is the non-critical section execution and service sites 2 to M are the different locks. The mean service time of server 1 is the mean time between locks on one processor. The mean service time of all the other servers is the mean critical section time for the corresponding lock. The probability $p_1 = 0$ and p_2 to p_m are given by the relative frequencies of use of the different locks. Since we have N processors, each of which is entering a critical section with rate μ_1, server 1 was made a load dependent server so that its service rate is $\mu_1 * n_1$ where n_1 is the number of customers at site 1.

The locks classified as 'lock on a page' are modelled as a multiserver. The number of subservers is adjusted till the predicted blocked time for these locks agrees with the measured block time. When we tried to model the 'Miscellaneous' locks as a multiserver, the model predicted more blocked time than what was observed for all the miscellaneous locks, even with the number of subservers equal to the number of processors. We therefore modeled the miscellaneous locks as M_{misc} locks in parallel such that each was attempted with equal probability and each has the same mean service time equal to the mean critical section time for miscellaneous locks as shown in figure 5.2. M_{misc} is adjusted until the predicted blocked time agrees with the observed blocked time.

The model is used to calculate the time lost due to blocking which can be interpreted as the processor power lost due to blocking. Consider the blocking at the i^{th} lock (server) for a moment. When two processors are present at the server, one is

actually receiving service and the other is waiting. When three processors are at the server, two processors are waiting.

Buzen gives a computationally efficient method for calculating the probability of k customers being present at the M^{th} server $P(n_M = k)$ in the load dependent server case.

$$P(n_M=k) = \frac{(X_M)^k * g(N-k, M-1)}{A_M(k) * G(N)}$$

Where, the X, A, G and g are the same as those defined in [BUZE73].

By permuting the numbers assigned to the locks, one can get the probability of n customers being present at the i^{th} server for $0 \le n \le N$ and $2 \le i \le M$.

Fraction of the time lost due to blocking at the i^{th} server is

$$lost_i = P(n_i=2) + P(n_i=3)*2 + P(n_i=4)*3 + \ldots + P(n_i=N)*(N-1). \quad (2 \le i \le M)$$

It then follows that,

Total fraction of time lost

due to blocking = $\sum_{2 \le i \le M} lost_i$

In order to validate the model, we have to calculate the blocked time as seen by one processor since the hardware monitor measures only one processor. Since all processors are assumed to be identical, this is just the total blocked time divided by the number of processors N.

Figure 5.3 Validation of the Central Server Model

	Program 1	Program 2	Program 3
Measurement			
Number of active processors (N)	13	14	12
Percent time lost on one processor at specific locks	measured [predicted]		
processor list lock	0.1249 [0.105]	0.4991 [0.5119]	0.1439 [0.0837]
feasible queue 1	0.0143 [0.0167]	0.2577 [0.1385]	0.0814 [0.0335]
feasible queue 8	0.0163 [0.0087]	0.0051 [0.0023]	
core lock	0.0152 [0.0212]		0.0698 [0.0637]
stop mail box	0.0066 [0.0177]		
Lock on a page	0.0697 [0.1182]		
KMPS lock	0.0023 [0.0032]		
Miscellaneous	0.0490 [0.0352]	0.0687 [0.1253]	0.4393 [0.4667]
Total percent time lost for one processor	0.298 [0.325]	0.831 [0.776]	0.735 [0.653]

Figure 5.4 Predictions of the Model

(a) Increasing number of processors:			
Total percent time lost on one processor for a			
32 processor system	0.9593	2.5863	2.1046
40 processor system	1.3011	3.9816	2.8040
48 processor system	1.7033	6.1505	3.586
(b) Using only one lock for all miscellaneous critical sections:			
Total percent time lost on one processor for a			
N* processor system	1.18	1.99	1.60
32 processor system	5.42	8.73	6.85
40 processor system	9.70	15.42	10.56
48 processor system	17.10	25.00	15.78
(c) Eliminating the processor list lock:			
Total percent time lost on one processor for a			
N* processor system	0.31	0.56	0.25
32 processor system	0.87	1.66	0.82
40 processor system	1.14	2.36	1.10
48 processor system	1.43	3.29	1.42

* N = 13, 14 and 12 respectively for the three programs.

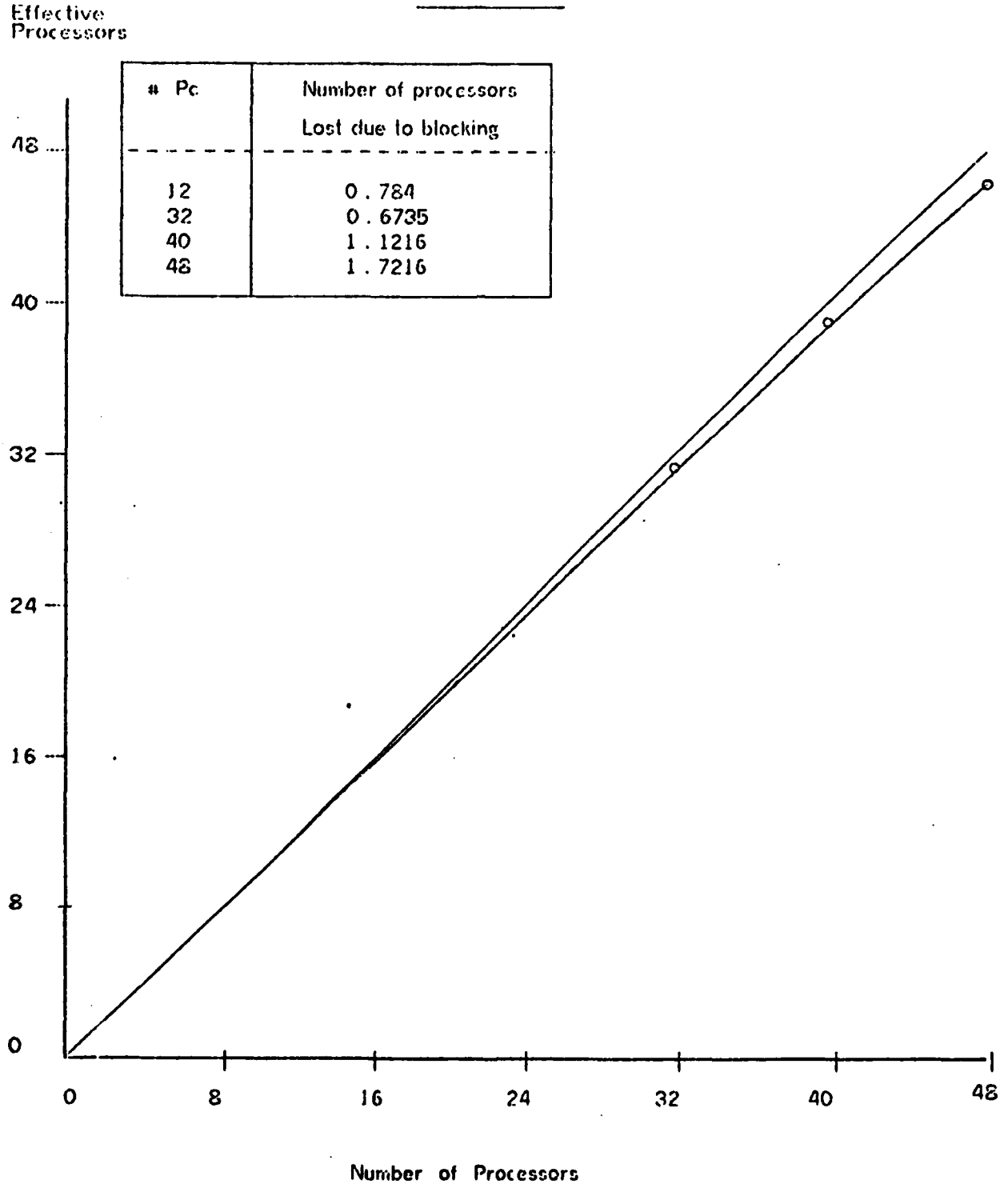
Figure 5.3 displays the measured and predicted percentage of block time. The

agreement between the measured and predicted values is fairly good for program 1 but for programs 2 and 3, the actual contention for the feasible queue 1 and the processor list is more than what is predicted. We do not yet know why the actual contention is so large. Programs 2 and 3 involve synchronization among many cooperating processes which become feasible almost simultaneously. This results in some amount of temporal correlation among processors when they attempt to access the feasible queues and the processor list.

The model can be easily extended for different numbers of processors (see figure 5.4 (a)). We assume that the number and characteristics of the locks do not change when the number of processors is altered. This might not be entirely valid since the miscellaneous locks will undoubtedly be more diverse with more processors and hence M_{misc} will have to be increased for better predictions. We decided not to alter M_{misc} however, since using the same M_{misc} will give us a worse case bound on the blocked time. It can be seen that Hydra has done a very good job of partitioning the shared objects into different critical sections. The processing power lost is very small even for the 48 processor case. Figure 5.5 displays the effects of the blocking for program 3.

We can also investigate the effects of reducing the number of locks in Hydra. If all the 'miscellaneous' critical sections in Hydra were to be executed using just one lock, considerable saving in storage will result since each object will not have to contain space for a lock. Our model predicts (see figure 5.4 (b)) that the performance penalty of this change will be small for a 16 processor system. For larger systems, it will still be advisable to have separate locks in each object. In the current implementation of Hydra, it is possible to achieve the necessary mutual exclusion without the use of

PROGRAM 3



# Pc	Number of processors Lost due to blocking
12	0.784
32	0.6735
40	1.1216
48	1.7216

Figure 5.5 Effects of Blocking for Program 3

'processor list' lock. In figure 5.4 (c) we display our predictions if this lock is eliminated. The time lost due to blocking will reduce as would be expected.

5.6. Conclusions

We have presented our measurements on the multiprocessor contention in accessing shared data. The measurements indicate that less than 1 percent time is lost due to blocking in Hydra. This negligible amount of degradation is a result of partitioning the shared data objects into small segments thereby reducing the critical section times. It should be noted that Hydra uses the locks for synchronization at only one of several levels. At higher levels in the system, semaphores and other message operations are used for synchronization which do result in context switch overhead but do not cause any loss of time due to blocking. From a purely performance point of view, Hydra could have used fewer locks with longer critical sections and it would still have had acceptable performance. This is an important result for the designers of future multiprocessor operating systems as well as for those trying to adapt a uniprocessor operating system to a multiprocessor.

We have also presented a central server model which predicts the observed blocking behavior reasonably well. The model is used to extrapolate the blocking behavior in systems with up to 48 processors. Even at 48 processors, the degradation due to lock contention appears to be small. Other interference problems at higher levels in the software [FULL76b] will be the limiting factors.

In any real multiprocessor operating system, the actual locking behavior will usually deviate from the simple central server model presented here. For example, some locks might always be executed in certain sequence or some locks may be nested inside

other locks. These situations might have to be modelled as a network of queues. Our study does not point out any major deviations from a central server model for locking behavior of Hydra, but the assumptions underlying our model will have to be verified for other operating systems.

6. Other Experiments Performed Using K.mon

In chapter 2 we presented an enumeration of the interesting parameters that need to be studied in a general purpose computer system (see figure 2.1). Many of these parameters are relevant for our evaluation of C.mmp and Hydra. However, it is not possible to investigate in depth all the performance parameters of a complex multiprocessor system like C.mmp in a short time. We performed many experiments in our study of these parameters but we discovered that we did not have all the special tools needed for an effective evaluation of a multiprocessor system. In this chapter, we present many specific experiments that were performed using the tools we had. It is easy to get bogged down in the details of experimental setup in such a case study. We have tried to avoid this problem by giving a brief description of the setup for each experiment and describing the goal of the experiment and the interpretation of the result in more depth. The experiments are discussed along the system levels presented in chapter 2. The system levels are :

1. Hardware architecture
2. Operating System design
3. Systems programming
4. Applications programming
5. Installation management

6.1. Measurements at the Hardware Architecture Level

K.mon is ideally suited for measurements at the hardware architecture level by virtue of the fact that it is capable of monitoring every cycle on the Unibus. Moreover, its sophisticated event detection mechanism can be used to select only the

interesting cycles, thereby removing the need for recording every Unibus cycle for post-processing. Most of our experiments at this level are directed at the evaluation of C.mmp. The measurements discussed here are:

1. study of memory interference in C.mmp
2. quantification of the effects of the small address space on Hydra
3. measurements of the types of memory accesses
4. a complete cycle by cycle trace

6.1.1 Memory Interference in C.mmp

C.mmp consists of up to 16 processors connected to as many as 16 memory ports via a cross-point switch. This arrangement leads to contention in the switch when two or more processors attempt to access one memory port at the same time. This problem has been studied earlier by Bhandarkar [BHAN73], McCredie [MCCR73] and by Baskett and Smith [BASK76] using analytical models. Our approach here is to actually measure the effects of contention in C.mmp when it is executing different workloads.

One large manufacturer estimates that for each additional processor in its multiprocessor system, 10 percent of the additional processing power is lost due to memory contention. The loss of processing power depends on many factors: the access and cycle times of the memory, the time taken by a processor to issue another main memory request after one is satisfied, the distribution of memory accesses to the different ports and the amount of I/O traffic [11]. For the overall system, these factors

---

[11] For our study of C.mmp, we could ignore the I/O traffic since it is not significant. However, if the processors are equipped with cache memories, the processor to memory traffic is reduced and the I/O traffic then becomes significant. The I/O traffic has a peculiar characteristic of accessing consecutive words and its effect needs to be considered explicitly especially for non-interleaved memories.

cannot be measured using K.mon since it can monitor only one processor at a time. Moreover, since the memory subsystem in C.mmp operates at a faster rate than the Unibus, we could not use K.mon (designed for a Unibus) to monitor the memory subsystem directly. We could only monitor secondary parameters like the length of a memory cycle as seen by a processor.

The average length of a cycle increases with the contention in the switch. Unfortunately, we could not measure the length of a cycle directly since K.mon is not equipped with a high speed clock necessary for such a measurement. Instead, K.mon is provided with six one-shot flip-flops which change their state at prespecified times (0.5, 1, 2, 4, 14 and 50 microseconds respectively) after a memory cycle is initiated. By examining the value of these flip-flops at the end of a memory cycle, we can determine the time bracket (or a bin) into which the length of that memory cycle falls. This in effect generates a crude histogram of memory cycle lengths and it gives an indication of the contention. Even though individual memory cycle lengths are not available, we can use the mean time value of a bin to approximate the average cycle length of all the cycles falling in that bin to yield the grand average of the time taken to complete a memory cycle.

Our experimental study attempts to quantify the extent of memory contention. We calculated the average cycle length for three different workloads:

1. Idle machine: This measurement was made as a basis for comparison with the other workloads.
2. XSEARCH: This is a root finding program which creates 16 cooperating processes executing in parallel. All these processes execute the same code but they all have individual copies of the code pages. Their activity is therefore distributed

throughout the memory ports. This workload is expected to produce approximately the same contention as many independent users executing different programs resulting in heavy use of most of the processors. The average cycle time is naturally larger than the previous workload (idle machine).

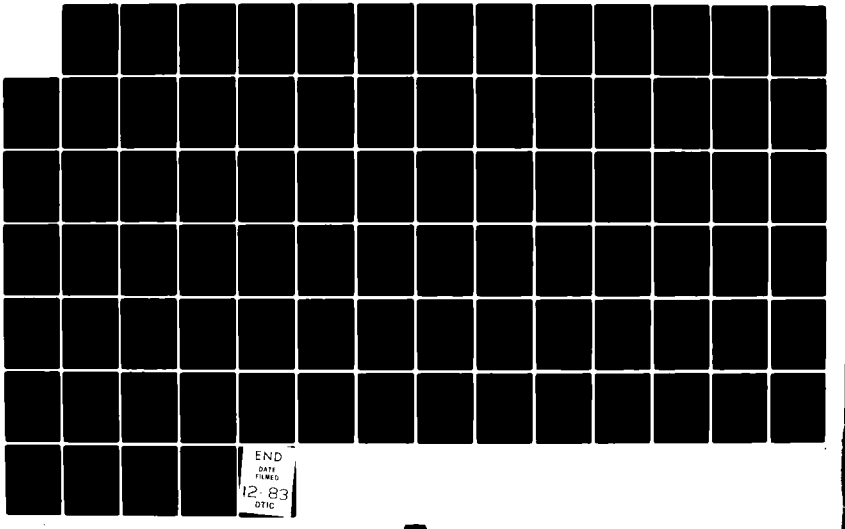
3. SEARCH: This is the same program as XSEARCH, except all the 16 processes share the code page, that is, they all make instruction fetches from the same memory port. A large amount of memory switch contention is to be expected with this workload.

We sampled 100,000 cycles at random for each of these three workloads. To simplify the experiment, K.mon was set up to measure the length of every memory cycle generated by the P.host. However, since K.mon's output rate is less than the main memory access rate of a PDP-11, the internal buffers of K.mon overflowed after collecting the cycle length for about 160 cycles. This gave rise to windows of measurement occuring after variable times thus effectively randomizing the measurements. The cycle length histograms produced by K.mon are given below:

AD-A134 925 PERFORMANCE EVALUATION AT THE HARDWARE ARCHITECTURE LEVEL AND THE OPERATI..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE M V MARATHE

UNCLASSIFIED DEC 77 F44620-73-C-0074 F/G 9/2 NL

2/2



END
DATE FILMED
12-83
DTIC

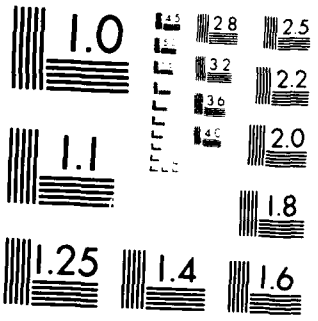


Figure 6.1 Length of a Memory Cycle

Cycle length microsec	Workload 1	Workload 2	Workload 3
0.0 - 0.5	0	0	0
0.5 - 1.0	88439	85134	69453
1 - 2	11404	13876	11601
2 - 5	71	958	3344
5 - 14	79	31	15421
14 - 50	7	1	181
above 50	0	0	0
Average length microseconds	0.8466	0.8834	2.335

It is interesting to note the significant tail for column 1 (the idle machine). Such a tail can arise only due to memory contention. In our data this effect is more pronounced because in C.mmp, memory conflicts are arbitrated according to strict hardware priority of the processors and our measurements were made on a processor having the lowest priority.

It can be seen that the contention for the second workload is quite small. This workload is expected to cause the processors to distribute their memory accesses uniformly across all the memory ports. Simple queueing models using a processor service time of 1.2 microseconds and a memory service time of 0.8466 microseconds (using the average from column 1), give the expected average waiting time when all the 16 processors are active to be about 1.2 microseconds. One explanation of the small increase in the average waiting time observed in our study is that the length of a cycle is composed of a fixed arbitration time, a fixed cable delay and a variable waiting

time for actual memory service. When two or more processors request service from a port, their arbitration times are overlapped and thus do not contribute to any lenghtening of their cycles. Another explanation is that in workload 2, each of the 16 processors is provided with independent instruction and data pages and it is possible that the whole system divides itself into 16 loosely coupled partitions leading to a low interference. Kmon cannot be used to verify these explanations since it is not capable of measuring memory arbitration time or the memory access behavior of all the processors at once. It is however, an interesting problem and we hope it will be studied in later investigations.

6.1.2 Small Address Space Problem on PDP-11

Since the PDP-11 has a small address space (64 K bytes), C.mmp uses a set of relocation registers to access its large physical memory. The virtual address space of a PDP-11 is divided into 8 pages of 8 K bytes each for this purpose. There are 8 relocation registers (henceforth called RR's) corresponding to these 8 pages. The RR's are used to translate a virtual address into a physical address [WULF72]. C.mmp utilizes 4 operating modes (kernel, user, I/O and unused) with 8 RR's each.

C.mmp thus provides an environment for writing large programs and gives us the unique opportunity of observing how the small address space of a PDP-11 affects the execution of large programs. In any large program, some time has to be devoted to maintaining the RR's in order to make different pages of the program accessible. The time spent doing this is clearly the price one has to pay for writing large programs to run on a small address space machine. Our experiment is designed to study and quantify this cost.

We used the Hydra system itself as our test program. Even though the cost of using

a machine with a small address space depends greatly on the manner in which the large physical memory is used [12], we believe that Hydra executing a reasonable workload is a good example of a large program using a small address space. Hydra consists of about 50 pages of instructions and data. It uses the kernel RR's to access these pages in the following manner:

Figure 6.2 Kernel mode Relocation Registers

RR	Function
0	fixed, stack page
1	fixed, common data page
2	overlay, data page
3	overlay, data page
4	overlay, instruction page
5	fixed, common instruction page
6	fixed, local memory
7	fixed, I/O page

Note that only RR4 is used to access the overlayed instruction pages and RR2 and RR3 are used for overlayed data pages.

The experiment is conducted in three parts, each giving successively detailed pictures. In the first part, all accesses[13] (read or write) to the Kernel RR's were counted over the duration of a second along with all cycles taking place on the machine, all kernel cycles and all kernel instructions. This gives a general estimate of the cost since any access to a RR is a direct result of an attempt to gain accessibility to a new page. In part 2, we trace individual accesses to RR2, RR3 and RR4 instead of

---

[12] For example, the cost is less if the memory is used to store large vectors and all words in a page are scanned before switching to another page.

[13] It is important not to confuse an access to a RR from the program for the purpose of inspecting or modifying it with an access from the hardware machine for the purpose of translating a virtual address. Here we are concerned with the former.

just counting them. This is used to see if the accesses to the RR's are uniformly distributed in time throughout the Kernel execution or bunched together. Moreover, since the data being read/written is also traced, it is possible to identify redundant writes , that is, when the same value is written back into a RR. Finally, in part 3, we trace each cycle taking place on P.host to determine how many instructions and cycles are actually associated with an access to a RR and to study in general the reasons behind changing the RR's.

Experimental Configuration:
Processor:C.mmp processor 3. (PDP-11/40).
Workload: Hydra executing XSEARCH (see section 6.1.1)

PART 1

Figure 6.3 The Rate of RR Accesses.

The counts for the following four quantities are provided for sixteen one second intervals:

All Kernel cycles	Kernel cycles	Kernel instructions	Accesses to RR's	Instructions per access	
353000		107645	45226	2913	15.52
362328		177341	73433	5130	14.31
340005		78579	33568	1843	18.21
350964		132763	55957	3438	16.27
308878		115049	48256	3099	15.57
338757		78218	33258	1837	18.10
320282		77544	33161	1798	18.44
344202		86569	36759	2072	17.74
312000		92972	39239	2359	16.63
401517		164851	68575	4729	14.50
335477		117706	49366	3196	15.44
358297		82876	35238	1980	17.79
371435		160413	66763	4595	14.52
347500		77400	32973	1824	18.07
372426		167739	69702	4768	14.61
400239		165853	68953	4653	14.81

The average number of instructions between RR accesses: 16.28

It can be seen that a RR is accessed on an average of every 16.28 Kernel instructions. Assuming one instruction per access, this corresponds to an overhead of

about 5.5 percent. It should be noted that this overhead considers only the execution time penalty. However, the small address space forces Hydra to use 32 bit addresses internally (giving the RR value and the displacement within the page) to access many routines and data objects. We expect the storage space overhead to be significant even though it did not form a part of our study. The next question is to determine if all the three variable RR's get accessed with the same frequency. This is done in part 2.

PART 2

Figure 6.4 The Accesses to Individual RR's.

	Cycle type	Relocation register	Kernel instructions since last access	
	Write	RR3		
	Write	RR3	11	
	Write	RR4	69	
	Write	RR2	2	
*	Write	RR2	13	
	Write	RR4	56	
	Read	RR3	31	
	Write	RR3	2	
	Write	RR3	9	
	Write	RR4	25	
*	Write	RR4	36	
	Write	RR4	37	
	Read	RR4	34	
	Read	RR4	24	
	Write	RR4	1	; exchange
	Read	RR2	23	
	Write	RR2	1	; exchange
	Read	RR4	9	
	Write	RR4	1	; exchange
	Write	RR3	33	
	Read-pause	RR3	6	
	Write	RR3	0	
	Write	RR4	20	
	Write	RR2	5	
*	Write	RR2	35	
	Read	RR2	7	
	Read	RR4	26	
	Write	RR2	6	
*	Write	RR2	34	
	Write	RR3	12	
	Read	RR3	4	
	Write	RR2	0	
	Read	RR2	8	
	Read	RR4	24	
	Read	RR4	24	
	Write	RR4	1	; exchange
	Write	RR3	44	
	Read	RR3	97	
	Write	RR4	13	
	Read	RR3	3	
	Read	RR2	3	
	Read	RR3	1	
	Read	RR4	3	
	Read	RR4	1	

* indicates a redundant write
Accesses to RR2: 12
Accesses to RR3: 14
Accesses to RR4: 17

Average kernel instructions between RR access: 18.05

It can be seen that the measurements on this level yield an average of about 18 Kernel instructions per RR access. Comparing it with 16.28 obtained in part 1, one can infer that the accesses to the RR's are distributed uniformly in time throughout the Kernel execution. Quite a few of the 'write' accesses are seen to be redundant . These arise because it is easier to write the required value in a RR than to check if the RR already has the same value. It is interesting to observe that the accesses are almost evenly spread across the three RR's. If one of these three was found to be lightly used, it might have been advisable to use it permanently to access the most heavily used overlayed page.

It is interesting to consider if the overhead of relocation register maintenance could have been reduced if an exchange instruction were available for the PDP-11. The purpose of this instruction would be to store the current value of a RR on the stack and to replace it with another value. In figure 6.4 we have marked all the RR accesses that could have been saved if such an exchange instruction were available. It can be seen that 4 out of the total 43 accesses could have been saved. This would have resulted in increasing the average number of kernel instructions between successive RR accesses to 19.9 instructions.

PART 3

We traced two different processors for this part to study the low level operations

leading to a RR access. Processor 3 on C.mmp has no I/O devices and the Kernel execution on it is limited to executing the Kernel calls. Processor 0 on the other hand, has many I/O devices and it executes many interrupt routines. The two processors indeed exhibit different traces. K.mon can trace only a small number of cycles before overflowing. This gives rise to a rather small window in the Kernel execution to make any general comments regarding why and how the RR's are accessed.Figure 6.5(a) displays a trace for processor 0 to illustrate the instruction sequences leading to a RR access. Figure 6.5(b) displays a part of a similar trace for processor 3.

The traces reveal two different kinds of overheads associated with maintaining the RR's. In figure 6.5(a), 10 instructions (6 through 14 and 21 through 22) are required to call a subroutine in an overlay page. Figure 6.5(b) shows that 3 instructions are required to gain accessibility to an overlay data page and one instruction is required to switch back to the previous data page. In parts 1 and 2 we only looked at the cost associated with RR accesses, that is, we did not explicitly consider the additional cost associated with calling a subroutine residing in an overlay page. The overall cost will depend on how many times such subroutines are called. We do not have sufficient data regarding that to draw any general conclusions. However, in appendix D we provide an execution trace of Hydra which shows that 2.5 times more instructions are executed in the overlay pages compared to the common code page. In the specific case of the program being monitored in part 2 above, we observe that 17 accesses were made to RR4 during 774 Kernel instructions. Since two accesses to RR4 are required to access an overlay instruction page and return to the previous page, the cost of the 17 RR4 accesses is about 85 instructions. Similarly, the cost of the accesses to RR2 and RR3 is 24 and 28 instructions respectively. The total cost of

maintaining RR's is thus 137 instructions out of 774 (about 18 percent). It should be noted that this is at best a rough estimate of the real cost involved in maintaining the RR's. The study of costs and benefits of using a 16 bit machine to write large programs is an interesting topic for future research.

Figure 6.5(a) Instruction Trace for Processor 0.

```
( 1) MOV R4,-(SP)                ;save value of R4
( 2) MOV R5,-(SP)                ;save value of R5
( 3) MOV R1,-(SP)                ;put value of R1 on the stack as a parameter
( 4) MOV 12(SP),R1               ;put parameter in R1
( 5) JSR PC,@(SP)+               ;jump to subroutine whose address is on stack
( 6)    MOV @#164064,-(SP)       ;save value in RR2 on the stack
( 7)    MOV @#164066,-(SP)       ;save value in RR3 on the stack
( 8)    SUB #12,SP               ;allocate 5 words on the stack
( 9)    MOV @R0,-(SP)            ;pass parameter to the $LINK
(10)    JSR R5,$LINK             ;jump to subroutine $LINK
(11)       MOV (R5)+,R0          ;move address of the routine to be called to R0
(12)       MOV @#164070,-(SP)    ;save the value of RR4 on the stack
(13)       MOV @(R5)+,@#164070   ;move the new value into RR4
(14)       JSR PC,@R0            ;jump to subroutine whose addres    in R0
(15)          MOVB 6(SP),R0      ;get parameter from the stack
(16)          BIC #177740,R0     ;clear unwanted bits
(17)          MOV 26332(R0),@#164064  ;move new value into RR2
(18)          MOV 6(SP),R0       ;prepare to return value of the ro     ) R0
(19)          BIC #20037,R0      ;returned value in R0
(20)          RTS PC             ;return from subroutine
(21)       MOV (SP)+,@#164070    ;pop back value of RR4 from stack
(22)       RTS R5                ;return from $LINK
(23)    MOV R0,R1                ;some useful work
(24)    CLR R2                   ;same
(25)    MOV @#164064,12(SP)      ;move value of RR2 into a local
(26)    MOV 30(R1),10(SP)*       ;move value into local
                                 ;address 30(R1) is 42130, which contains #76100
                                 ;NOTE: address 42130 uses new value in RR2
(27)    MOV 10(SP),@SP           ;move local to top of stack
(28)    JSR R5,$LINK             ;jump to $LINK
(29)       MOV (R5)+,R0          ;same sequence as above
(30)       MOV @#164070,-(SP)
(31)       MOV @(R5)+,@#164070
(32)       JSR PC,@R0
(33)          MOVB 6(SP),R0      ; and so on.....
(34)          BIC #177740,R0
(36)          MOV 26332(R0),@#164064
(37)          MOV 6(SP),R0
(38)          BIC #20037,R0
(39)          RTS PC
(40)       MOV (SP)+,@#164070
(41)       RTS R5
```

Figure 6.5(b) Instruction Trace for Processor 3.

```
(1)  MOV @R3,14(R2)          ;not relevant
(2)  MOV @#164064,(SP)       ;save the old value of RR2 on the stack
(3)  MOV @R4,R2              ;code to find the new value of RR2
(4)  MOV 10(R2),@#164064     ;load the new value in RR2
 ....
(19) MOV @SP,@#164064        ;load the saved value of RR2
```

6.1.3 Study of memory access types

On the PDP-11, memory words can be accessed in one of four access modes: read, read-pause, write and write-byte. It is interesting to investigate the relative frequency of the use of these modes. This measurement is particularly easy with Kmon, since it has four counters which can be programmed to count the occurrences of these four modes. Figure 6.6 presents the frequencies of the use of these modes on different models of PDP-11 executing the same program. A PDP-11 model 20 needs two memory cycles (a read-pause followed by a write) to write a value in its main memory. PDP-11 model 40 , on the other hand, needs only the write cycle. Both models, however, use the read-pause cycle for instructions like increment and decrement. The difference in the use of the read-pause cycle is quite evident in the figure. One of the main uses of this information is in the design of cache memories. C.mmp can benefit from the introduction of a cache memory for each processor. However, because C.mmp is a multi-processor, the cache can contain only read-only words. The percentage of read cycles is thus an important factor in determining the value (i.e. the hit ratio) of such a cache memory.

Workload: Hydra executing XSEARCH (see section 6.1.1).
The counts are made over one second duration. The values for ten typical seconds are provided below for the two processors.

Figure 6.6 (a) Processor 3 (PDP 11/40)

	Read percent	Read-pause percent	Write percent	Write-byte percent	Total cycles
	85.61	2.83	10.76	0.80	308217
	85.04	2.70	11.44	0.81	350160
	84.97	2.69	11.53	0.81	333973
	85.48	2.84	10.88	0.80	323567
	85.58	2.79	10.82	0.80	291352
	85.59	2.69	10.94	0.77	350794
	85.11	2.71	11.37	0.81	355535
	85.14	2.69	11.35	0.80	326851
	85.59	2.79	10.82	0.80	286232
	85.09	2.72	11.37	0.81	363985
Mean	85.32	2.745	11.128	0.801	

Figure 6.6 (b) Processor 0 (PDP 11/20)

	Read percent	Read-pause percent	Write percent	Write-byte percent	Total cycles
	79.25	9.11	11.08	0.56	247756
	79.18	9.12	11.16	0.54	253720
	78.97	9.25	11.23	0.56	242460
	78.60	9.40	11.47	0.56	240812
	79.19	9.12	11.16	0.53	238226
	79.02	9.19	11.26	0.52	227607
	78.92	9.24	11.33	0.51	248309
	79.06	9.18	11.21	0.53	239657
	79.03	9.22	11.17	0.58	244516
	78.75	9.32	11.42	0.50	232767
Mean	78.997	9.215	11.243	0.536	

6.1.4 Comprehensive unibus cycle trace

This is one of the traditional measurements which is capable of answering many

questions regarding the hardware architecture. Such a trace has been described by Bordeen [BORD71] for the Univac 1108. To gather such a trace, the hardware monitor has to possess a very high bandwidth output device such as a fixed head drum or a large amount of core memory. Unfortunately, Kmon lacks such features and is thus not very useful for this measurement. It is however, possible to record 160 consecutive unibus cycles before the internal buffers in Kmon overflow. We could therefore gather a trace consisting of windows of 160 consecutive cycles which could still provide many of the answers. The trace was post-processed to yield the instruction mix, frequently occuring instruction sequences, frequency of use of all combinations of mode and register pairs, a histogram of the number of memory cycles per instruction, a histogram of index values off the stack register and a histogram of immediate mode operands. Because of the limitation of gathering only 160 consecutive cycles, we could not study the branch distance behavior. Appendix C presents the results of this study.

It should be noted that it was not possible to obtain a trace consisting of more than a few thousand instructions. We could not perform this measurement on more than a few systems. The results are not very general and do not compare very well with those obtained using other methods. For example, appendix C displays the instruction mix obtained using the trace. It shows that the instruction 'MOV' was used only 16 percent of the time in the traced instructions. This does not agree with the value of 31.2 percent obtained in chapter 4. The discrepancy arises because, appendix C reports the results for only 12000 instructions from one specific program. The rest of appendix C should also be used with caution due to the same reason.

Future research should concentrate on obtaining longer traces of many different programs in order to draw meaningful conclusions. It will then be possible to answer

many fundamental questions regarding the PDP 11 architecture and it will also help in the design of new architectures. For example, it is interesting to determine the utility of providing immediate operands (3 or 4 bits long) in PDP 11 instructions. Appendix C displays how many small immediate mode operands are used in the program under study. Similarly, the histogram of index values off the stack pointer presents how formal parameters passed on the stack are accessed by the program. Another measurement (not presented here) is to examine how often a 'MOV' instruction uses addressing mode 0 for its source or destination operands. In all such cases single operand LOAD or STORE instructions would have been sufficient. There are many such questions that can be answered once the traces are obtained. The statistical experiment presented in chapter 4 can be used to quantify the variance of the measured quantities.

6.2. Operating System Design Level

As discussed in section 2.2, a hardware monitor is a useful tool for measurements at this level also. Many measurements can be performed without altering the operating system, but the task is simplified if the operating system can be modified to supply certain hard-to-obtain parameters to the hardware monitor. We will present only three of the many measurements performed using Kmon since these are more generally applicable.

1. the execution profile
2. study of processor hardware priority changes
3. functional trace of the operating system

6.2.1 The execution profile

Execution profile refers to the measurement of the frequency of execution of different regions in a program. The technique used is to sample the program counter at random and output its value. This generates a list of absolute addresses which by itself cannot be easily interpreted by operating system programmers. This list is therefore post-processed using a map [14] of the program to generate the execution profile. It is then possible to optimize the heavily used portions of the program or to alter the algorithms used.

There are two problems that have to be solved before such a measurement becomes feasible. One problem is at the input level, where the hardware monitor has to select the program counter values belonging to the operating system. This can usually be done by using address comparators to isolate the operating system region from the machine's address space. The operating system can also provide this information to the hardware monitor using some signalling mechanism. The second problem is at the post-processing level, when the mapping between the operating system's routine names and their absolute addresses cannot be determined because portions of the operating system are dynamically relocatable and / or overlayed. There are no general solutions to this problem, except that the operating system can be programmed to supply the new overlay number when it is brought in. For Hydra, we look at the value being written into RR4 (see section 6.1.2) to determine which overlay page is being used.

---

[14] a map associates names of the routines in the program with their absolute addresses

The execution profile for Hydra is shown in appendix D. This measurement helps in deciding which routines should be in the resident part and which should be in the overlay part. It has also helped in selecting routines for optimization and for implementation in microcode. It can be seen that considerable time is being spent in the register save/ restore routines. These are natural candidates for implementation in microcode. Our analysis program also provides a list of routines ordered according to the number of instruction samples falling in them. It can be seen that many of the commonly used routines (e.g. ENQ , SELCTE and REQPRO in page 44, and MKRN.C in page 23) can be moved to the fixed code page thereby avoiding changing RR4 every time they are called.

6.2.2 Changes in the processor hardware priority

The PDP-11 has eight processor priority levels. Execution at any of these levels can only be interrupted by an interrupt occuring at a higher priority. Hydra uses a convention that the user programs can execute at processor levels from 0 through 3 and the operating system executes at levels 4 through 7. Different devices cause interrupts at different levels from 4 through 7. Since device interrupts have to be serviced within a short time of their occurrence, it is necessary to restrict the operating system execution at high priority levels. K.mon was used to detect high priority executions exceeding a certain threshold time. Since processor priority is not available as a signal on the unibus, special probes were connected to the the priority bits in the processor. K.mon was programmed to detect the changes in priority level and record the time at which the change occured and the address of the first instruction following the change. The supervisory program on P.sup detected the executions exceeding the threshold.

Sample output of this experiment is provided below in figure 6.7. The address of the instruction (including the page number for addresses in the overlayed instruction pages) following the rise in the priority level is given so that corrective action can be directed at the proper place in the operating system. In figure 6.7, all the high priority execution was caused by device interrupts and so we directly report the device causing the interrupt in stead of giving the address of the interrupt routine. The following measurement was performed to detect the high priority execution exceeding one millisecond.

Figure 6.7 Changes in Processor Hardware Priority

Priority	Page	Device	How long (microsec)
6	local	Console TTY	3675
6	local	Line clock	2951
6	local	Console TTY	3672
6	local	Line clock	2822
6	local	Console TTY	4951

6.2.3 Functional trace of an operating system

The purpose of this measurement [15] was to measure the CPU and I/O processing characteristics of RSX11-M for use as input to a simulation model of this operating system. A set of major processing functions were identified for use in the model. These were:

Terminal input handling

task activation

task initiation (with and without checkpointing)

---

[15] This experiment was performed jointly by Dr. Dermot Bredin of Digital Equipment Corp and the author

task execution (a Fortran program with subroutine calls, disk I/O and an overlay structure)

task termination

terminal output handling

A number of places in the operating system code were then identified so that a trace of execution at these places is sufficient to give the time required to perform the above functions. One word in the operating system's data area was designated as the 'hook' word and code was introduced at these places to write a value in the hook location to uniquely identify the place. K.mon was set up to detect any 'write' operation into the hook word and record the value written and the time stamp. All the commands given to the disks were also monitored separately by tracing all the 'write' operations into the device registers (see section 6.5 for another experiment performed by monitoring the device registers). The output of the monitor was then analysed to give a complete trace of activities of the operating system and the disks. It was necessary to restrict the execution to a single user execution a simple program in order to interpret the trace successfully. A small part of the trace is included in Appendix E. The information obtained with this measurement can also be obtained tracing the events in software. The only reason K.mon was used for this measurement was to introduce negligible perturbation in the operation of the operating system.

6.3. Systems Programming Level

This level includes the compilers and their run-time systems, the utility programs and the file systems. The execution profile (see previous section) is again the most important parameter at this level. This level is characterised by a strong interaction

with the operating system. It is therefore interesting to monitor the service calls to the operating system. On the PDP-11, a service call is initiated by executing a special instruction (EMT or TRAP) and the operating system returns to the user also using a special instruction (RTI or RTT). K.mon was programmed to monitor the execution of these instructions and record the type of call and the time stamp. This yields the frequency of use of the different service calls and the histogram of the execution time for each. In some cases, the service calls to the operating system consume a significant amount of the total execution time of a systems program. It is sometimes possible to alter an algorithm to eliminate certain service calls and hence a measurement was designed to give the total time consumed by each call in a Fortran compiler.

Table 6.8 lists the instruction addresses in the Fortran Compiler from where an operating system service call is made. An instruction address is uniquely identified by the pair (overlay number, address). For each instruction address, the maximum time spent in the operating system to complete the call from that address is given along with the total time spent in the operating system due to a call at that address. The total number of times a call at a particular address is executed is also given to guide the optimization of compiler algorithms.

The measurements were made for the entire duration of the compilation for a typical user program. The Fortran compiler was the only program running on the computer when the measurements were made.

Figure 6.8 Time Consumed by Calls from a Fortran Compiler

Total time of measurement: 21636 milliseconds
Total time inside the compiler: 9114 milliseconds = 42.1 percent

Overlay number	address (octal)	maximum time microsec	total time microsec	number of calls	Description
root	6010	2807	514122	226	QIO from the overlay handler
root	6036	31569	3846073	227	QIO from the overlay handler
root	10502	127928	2148581	265	QIO from the file system
root	10640	140891	5735124	207	File system: Wait for I/O completion
root	10652	2256	114014	205	File system: Wait for I/O completion
0	22466	648	5816	9	File initialization
0	27272	1570	39625	39	Assign logical unit number (similar to channel number)
0	27406	1491	40328	40	Assign logical unit number
0	17024	662	8565	13	Get task parameters
0	17032	881	8711	13	Get task parameters
0	17114	565	7341	13	Set software trap vectors
0	17754	881	8038	12	Get time parameters
0	20230	375	8636	13	Get task parameters
2	13640	1127	13021	14	Assign logical unit number (for the file system)
2	13754	1202	14031	14	Assign logical unit number (for the file system)

This measurement can also be conducted in software, but a hardware monitor can gather this information independent of the operating system as long as the same instructions are used for call entry and exit.

In order to help reduce the overall time required for compilation, the following experiment was performed. K.mon was set up to monitor three states while the compiler was run stand-alone on the machine:

1. Compiler execution
2. Operating system execution
3. Wait for a device (usually a disk) to complete transfer

The output of the experiment includes the percentage of time spent in each of the three states. A large waiting time indicates poor overlap structure of the compiler. Even though in a multi-programming system, a large waiting time does not necessarily cause decreased throughput, it does increase the response time experiencedby the users. This measurement can be used continually to quantify the improvement (or otherwise) in the execution of the systems program as modifications are made in the program or in the operating system algorithms. Figure 6.9 presents our results for the Fortran compiler. It can be seen that the compiler exhibits a poor overlap structure spending over a third of the elapsed time waiting for the I/O completion.

Figure 6.9 Fortran Compiler: Overlap Structure

Total time of measurement: 33380 milliseconds
Total time in the User state: 18253 milliseconds = 54.68 percent
Total time in the non-user state: 15007 milliseconds = 44.96 percent
Breakdown on non-user time:
Total I/O wait time: 11544 milliseconds = 34.58 percent
Total operating system overhead: 1723 milliseconds = 5.16 percent
Total file system overhead: 1619 milliseconds = 4.87 percent

6.4. Applications Programming Level

K.mon is designed for PDP-11, which is a mini-computer and consequently, we were not able to apply it to any large installation supporting many applications programs. We therefore have limited experience in the applicability of a hardware monitor at this level. Instruction execution profile remains the most important parameter even at this level. The problems mentioned in section 6.2.1 are compounded by the fact that the execution profile at the high level language statement level is required by the

applications programmers and this is very difficult to obtain using a hardware monitor. There are many ways to get around these problems. In our opinion, the use of a hardware monitor for this measurement becomes clumsy and not generally applicable. The best solution will be to provide the necessary facilities in the compilers and operating systems.

6.5. Installation Management Level

As mentioned in the previous section, K.mon has not been applied to any large installation supporting many users, for which installation management is necessary. The main performance parameter at this level is the equipment utilization and overlap. We therefore designed an experiment to measure the overlap between the processor and any device (we have restricted our attention to disks and drums only). We did not consider number of jobs per day or the average CPU utilization as meaningful measurements with K.mon.

Since devices are not accessed through channels on the PDP-11, we experienced one simplification and one problem. The simplification is that separate probes are not needed to monitor the channel activity. The device registers are accessed through the unibus and so they can be monitored with the address comparators in K.mon. The problem arises in the definition of 'overlap'. In conventional machines, the channel busy and processor busy signals can be AND'ed together to detect overlap. We defined overlap as the number of processor cycles between issuing the start read/write command to a device and receiving the completion interrupt from the device. Clearly, if the processor executes a WAIT instruction immediately after giving the start command to the device, the overlap will be zero.

K.mon is set up as follows:

Event 0: 1 microsecond clock

Event 1: 'write' into any device register. When detected, outputs time stamp, register address, value being written, values of counters 3 and 4.

Event 2: fetch of an interrupt vector. When detected, outputs time stamp, interrupt vector address(this identifies the device causing the interrupt), values of counters 3 and 4.

Event 3: counts in counter 3 the number of unibus cycles initiated by the processor.

Event 4: counts in counter 4 the number of all unibus cycles.

The trace is analysed by a program which interprets the commands given to the various devices and effectively reconstructs the device activity. It can, for example, find out which cylinder of a disk was accessed. The previous position of the disk arm is available to the analysis program from its interpretation of the previous command, hence it can determine the magnitude of the disk arm movement for every command. It can also determine the number of words transfered with every command, the time taken to complete seeks and transfers and the utilization of the different units of a device. Overlap is the difference between the values of counter 3 between event 1 ('go' command being given to the device) and the following event 2 (interrupt). When a device receives a read/write command, it cannot accept any other command until the transfer initiated by the first command is complete. So the utilization of a device (similar to the utilization of a channel) can be defined as the total time a device was busy divided by the total time of measurement.

Appendix F contains a sample output of this experiment. Because of the low output

bandwidth of Kmon, the trace consists of many windows from the actual execution of the system. This could have been avoided by using a hybrid scheme where the timing and overlap data were obtained with Kmon and the remaining data obtained by inserting suitable measurement code in the operating system. It is however advisable to perform the experiment without modifying the operating system, since then the same experiment can then be used for measurements on any operating system.

7. Conclusions and Further Research

In this dissertation we have concentrated on the measurement and analysis problem of computer systems at the hardware architecture and the operating system kernel design levels. In chapter 2 we examined the performance parameters at various system levels and discussed the applicable measurement tools. Since our interest lies in the phenomena covering the range of a few instructions, the most appropriate measurement tool for our purpose was a hardware monitor. A hardware monitor, however, is a versatile tool applicable to other system levels as well, so some effort was devoted in studying the hardware monitoring techniques in general. A brief description of our hardware monitor Kmon was presented in chapter 3.

Chapter 4 discussed a major experiment designed to address the question of the variability of the instruction mix. The experiment was designed to quantify the variance caused by 3 factors and to enable comparison of their effects. Application of statistical experimental design methodologies is relatively new in the field of performance evaluation. We hope our success will trigger more interest in the scientific design of experiments in this field. Our measurements indicate that a statistically significant variation in the instruction mix is caused both by the application area and the program in a given area but not by the different phases of execution in the same program. It is therefore not advisable to attempt to over-optimize a processor for a particular application area. We also quantified an intuitively well understood fact that all the addressing modes of the PDP-11 are not equally useful. For double operand instructions , mode 5 (auto-decrement deferred) is almost never used. Many of the instructions were also shown to be seldom used. These results are important for the design, implementation or emulation of PDP-11 or similar processors.

Here too, as in any other inquiry, answers to one set of questions give rise to new questions. For example, two of our application areas use high level languages (Fortran and Cobol). It would be interesting to investigate the variance due to the use of different compilers. In our study, we have ignored this effect by assuming that similar machine instructions will be used to accomplish the type and amount of real 'work' being demanded by a high level language statement, independent of the compiler used. But this assumption certainly needs to be investigated. It will also be interesting to perform a similar experiment on a larger machine for which Cobol compilers have been available for many years and which possess Cobol-specific machine instructions.

It is interesting to look at the whole experiment in the light of the workload characterization problem. Intuitively the different application areas represent different workloads since it is clear that each of these areas is doing a different kind of 'work'. The Fortran programs are manipulating numbers for the purpose of solving equations, the operating systems are performing the processor and memory scheduling functions whereas the real time systems are responding to the events happenning in their environments. Our experiment is an attempt to characterize these intuitively different workloads in terms of their instruction mixes. It turns out that a meaningful characterization at such a low level is not possible due to the variation in the programs belonging to these areas. This negative result should not be interpreted as saying that a characterization at a higher level is not possible; in fact, future research should concentrate on the next higher level of atomic 'work' e.g. in terms of manipulations of higher level data structures like vectors, lists, process control blocks, and strings. Chapter 5 analysed the problem of software lockout for multi-processor operating systems. In order to maintain system integrity, certain shared objects have

to be accessed by only one processor at a time. Such a mutual exclusion gives rise to critical sections of code which can be executed by only one processor at a time. This results in a loss of time if a processor has to wait to access a shared data object until it becomes free. We showed that in Hydra, only three parameters control the time lost due to software lockout: the average lengths of the critical sections, the relative frequencies of use of the various shared data objects and the number of processors in the system. In other systems, the critical sections might be nested inside one another. A different model will have to be evolved in these cases. We observed that Hydra has been quite successful with respect to the software lockout problem; less than 1 percent of time was lost due to lockout even for a fairly busy system. Hydra contains many shared objects but the critical section times have been kept small resulting in smaller lost time. It might be interesting to consider other designs where fewer and larger critical sections are used with (perhaps) some saving in complexity or time without paying too much penalty in lost time.

Chapter 6 discussed other applications of K.mon. K.mon's limitations become apparent in the study of the memory contention problem and also in obtaining a complete memory cycle trace. Future hardware monitors should have provisions for measuring memory cycle times and for tracing each machine cycle. It can be seen, however, that a hardware monitor is applicable at all the system levels and this fact should be kept in mind when designing future hardware monitors.

References

A bibliography on Hardware Monitors is also included

AGAJ75 Agajanian A H. "A Bibliography on Systems Performance Evaluation" IEEE Computer vol 8 no 11. November 1975

AMIO72 Amiot L, Natarajan N K and Aschenbrenner R A. "Evaluating a Remote Batch Processing System" IEEE Computer Sept/Oct 1972. pp 24

ANDE74 Anderson V L and McLan R A. "Design of Experiments: A Realistic Approach" Marcel Dekker, Inc. New York 1974.

ARBU66 Arbuckle R A "Computer Analysis and Throughput Evaluation" Computers and Automation, January 1966.

ARND72 Arndt F R and Oliver G M. "Hardware Monitoring of a Real-time Computer System" IEEE Computer vol 5 no 4 July/Aug 1972. pp 25-29

ASCH71 Aschenbrenner R A, Amiot L and Natarajan N K. "The Neurotron Monitor System" AFIPS FJCC 1971 vol 39. pp 31-37

BASK76 Baskett F and Smith A J. "Interference in Multiprocessor Computer Systems with Interleaved Memory", CACM 19,6 pp 327-334, June 1976.

BHAN73 Bhandarkar D P. "Models of Memory Interference" PhD Thesis. Dept of Electrical Engineering, carnegie-Mellon University, 1973

BONN69 Bonner J R. "Using System Monitor Output to Improve Performance" IBM Systems Journal #4, 1969. pp 290-298

BORD71 Bordsen D T. "Univac 1108 Hardware Instrumentation System" ACM Sigops workshop on Systems Performance Evaluation. April 1971

BUCK76 Buck D. "A System for Controlling Remotely Programmable Hardware Monitors" University of Waterloo CCNG report 1-28 Oct 1976.

BUZE73 Buzen J P. "Computational algorithms for closed queueing networks with exponential servers" CACM vol 16, no. 9. Sept 1973 pp 527-531

CARL71 Carlson G. "A User's View of Hardware Performance Monitors" IFIP Congress 1971. TA-5 pp 128-132.

COCK71 Cockrum J S and Crockett E D. "Interpreting the results of a Hardware Systems Monitor" AFIPS SJCC 1971. vol 38. pp 23-38

CONN70 Conners W D, Mercer V S and Sorlini T A. "S/360 Instruction Usage Distribution" Report IBM-SDS TR 00.2025, May 8, 1970.

COLL76 Collins J P. "Performance Improvement of the CP-V loader through use of the ADAM Hardware Monitor ". Performance Evaluation Review ACM Sigmetrics vol 5 no 2, pp 63-67 April 1976.

DEC71 Digital Equipment Corp. Maynard, Mass. "PDP 11/40 Processor Handbook".

DYNA76 Dynaprobe 8016. Comten Inc. Two Research Court, Rockville Maryland 20850

ESTR67 Estrin G et al. "The Snuper Computer" AFIPS SJCC vol 30 1967. pp 645-656.

FOST71 Foster C G, Gonter R H and Riseman E M. "Measures of op-cod Utilization" IEEE Transactions on Computers 20,5 pp 582-584 May 1971.

FRYE73 Fryer R E " Memory Bus Monitor- A New Device for Developing Real Time Systems". National Computer Conf. 1973 pp 75-79. FULL73 Fuller S H, Swan R J and Wulf W A. "Instrumentation of C.mmp" Proc IEEE International Comp Soc Conf Feb 1973, pp 173-176

FULL76 Fuller S H. "Price/Performance Comparison of C.mmp and the PDP-10" Proc Third Annual Symposium on Computer Architecture, pp 195-202, January 1976.

FULL76b Fuller S H and Oleinick P N. "Initial Measurements of Parallel Programs on a Multi-mini-processor" IEEE CompCon pp 358-363, 1976.

GIBS70 Gibson J C. "The Gibson Mix" Report TR 00.2043, IBM Systems Development Division, Poughkeepsie, NY, 1970 (Research done in 1959).

GONT69 Gonter R H "Comparison of Gibson Mix with UMASS Mix" Publication number TM/RCC/004, University of Massachusetts, Research Computing Center.

HART70 Hart E L. "The User's Guide to Evaluation Products" Datamation Dec 1970. pp 32-35.

HUGH73 Hughes J. "performance Evaluation Techniques and System Reliability-A Practical Approach" ACM/NBS Performance Evaluation Workshop, March 1973.

HUGH74 Hughes J and Cronshaw D " On using a Hardware Monitor as an Intelligent Peripheral" Performance Evaluation Review, ACM, December 74.

HUGH76 Hughes J. "A Functional Instruction Mix and Some Related Topics" Proc. International Symposium on Computer Performance modelling, Measurement and Evaluation, Harvard Univ. March 1976. pp 145-153.

JOHN70 Johnson R R. "Needed: A Measure for Measure" Datamation Dec 1970. pp 22-30.

IBM63 IBM 1963. "Throughput Evaluations.... IBM 7090|7094 Data Processing Systems"

KOLA77 Kolanko R. "A Structured approach to Performance Measurement of Computer Systems" PhD thesis University of Waterloo. CCNG E-62. June 1977.

LIND76 Lindsay D S. "A Hardware monitor study of a CDC Kronos system" Proc.

International Symposium on Computer Performance modelling, Measurement and Evaluation, Harvard Univ. March 1976. pp 136-144.

LUCA71 Lucas H C. "Performance Evaluation and Monitoring" Computing Surveys vol 3 no 3 pp 79-92. Sept 1971.

LUND74 Lunde Amund. "Evaluation of Instruction Set Processor Architecture by Program Tracing" PhD Thesis. Department of Computer Science, Carnegie-Mellon University, 1974.

MADN68 Madnick S W. "Multi-processor Software Lockout", Proceedings 1968 ACM National Conference, pp 19-24.

MCCR73 McCredie J W. "Analytic Models of time-shared computing systems: new results, validations and uses". PhD Thesis. Computer Science Department Carnegie-Mellon University, 1973.

MORG73 Morgan D et al. "A Computer Network Monitoring System" IEEE Transactions on Software Engineering, Vol SE-1, no 3, Sept 73. pp 299-311.

NOE74 Noe J D. "Acquiring and Using a Hardware Monitor" Datamation vol 20 no 4 pp 89-95 April 1974.

NUTT75 Nutt G J. "Tutorial: Computer System Monitors" IEEE Computer vol 8 no 11. pp 51-61 November 1975.

PART76 Partridge D R and Card R E. "Hardware Monitoring of Real-time Aerospace Computer System" Proc. International Symposium on Computer Performance modelling, Measurement and Evaluation, Harvard Univ. March 1976. pp 85-101.

RAIC66 Raichelson E and Collins G A. "A Method of Comparing the Internal Operating Speeds of Computers", CACM 7,5 pp 309-310 , May 1966.

ROCK69 Rock D J and Emerson W C. "A Hardware Instrumentation Approach to Evaluation of Large Scale Systems" Proceedings of the ACM National Conference pp 351-367. 1969.

RUDD72 Rudd R J. "CPM-X A Systems Approach to Performance Measurement" AFIPS FJCC 1972. vol 41 part II.

SCHR71 Schroeder M D. "Performance of the GE-645 Associative Memory while Multics is in Operation" ACM Sigops Workshop on Performance Evaluation. April 1971. pp 227.

SEBA74 Sebastian P R. "HEMI (Hybrid Events Monitoring Instrument)" ACM Performance Evaluation Review, vol 3 no 4. 1974 pp 127

SNED67 Snedecor G W and Cochran W G. "Statistical Methods". The Iowa State University Press, Ames Iowa. 6 th edition, 1967.

STON75 Stone S H (editor) "Introduction to Computer Architecture". Science Research Associates, 1975.

SVOB73 Svobodova L. "Online Systems Performance Measurement with Software and Hybrid Monitors" ACM fourth Symposium on Operating Systems Principles pp 45-53. Oct 1973.

SVOB74 Svobodova L. "Computer Systems Performance Measurement: Instructions Set Processor Level and Microcode Level" Report AD/A000 946/4ST Stanford University 1974.

SVOB76a Svobodova L and Mattson R "The Role of Emulation in Performance Measurement and Evaluation" Proc. International Symposium on Computer Performance modelling, Measurement and Evaluation, Harvard Univ. March 1976. pp 126-135.

SVOB76b Svobodova L. "Computer Performance Measurement and Evaluation Methods: Analysis and Applications" Elsevier Computer Science Library 1976.

SWAN76 San R J. "Kmon- the Cmmp Hardware Monitor. A Proammers Manual" Department of Computer Science, Carnegie-Mellon University. Internal Report, 1976.

TABK72 Tabke R B. "Channel Monitor" IBM Technical Disclosure Bull. vol 15 no 4. 1395(1972) CCA8-7330.

TESD76 Tesdata MS. Tesdata Inc. 7900 Westpark drive, McLean Virginia 22101.

VOOR75 Voorhies D A. "The Hybrid Program Measurement Device: Design and Capabilities" MITRE Corp. report MTR- 3105 August 1975.

WIND73 Winder R O. "A Database for Computer Performance Evaluation " IEEE Computer vol 6 no 3 March 1973. pp 25-29

WULF75 Wulf et al. "The Hydra Operating System" Fifth ACM SIGOPS Symposium on operating systems Principles, Nov 1975.

Appendix A: Survey of Hardware Monitoring Techniques

A.1. Introduction

A list of performance parameters at various system levels and the various experiments performed using hardware monitors to gather these parameters were discussed in the previous chapter. This chapter surveys the hardware monitoring techniques that have been used in the past to perform these measurements. The study of *hardware monitoring systems* is broken down into three dimensions:

1. the event detection mechanism
2. the event response specification
3. the display mechanism.

Hardware monitoring systems from the initial IBM 7090 monitors to the current state of the art monitors are discussed along these dimensions.

Performance monitors for computer systems are available in many forms, often designed to measure very different parameters of an operational system. Performance monitors can be broadly classified into two types. First, the hardware monitors capable of sensing bits and words of the computer system's status. Second, the software monitors capable of interrogating software structures such as queues and job tables . More recently hybrid monitors have been used. These are hardware monitors assisted by software on the measured system to obtain information which is not available or is difficult to get for a pure hardware monitor.

Various computer professionals have different reasons for initiating measurement of a computer system. These include gaining understanding of the dynamic behavior of a system, observation and prediction of the effects of hardware and software changes,

obtaining parameters for analytic or simulation modelling and model validation. The performance parameters at various levels in a computer system are also different. Figure 2.1 attempts to characterize the performance parameters at various system levels and suggests the most valuable performance measurement tool for each level. Since the performance parameters at any level depend on the parameters of levels below it, it is possible to measure parameters at any level using tools most applicable to lower levels. The hardware monitor therefore, is a useful measurement tool at all system levels and an indispensable tool at the hardware architecture and operating system kernel design level. Earlier hardware monitors were restricted to low system levels but most new commercial monitors are geared towards the installation manager and applications programmer levels. This fact should be kept in mind when comparing past and present commercial hardware monitors.

A.2. Functional Components of a Hardware Monitor

The hardware monitors have evolved over time from simple summary devices for the IBM 7090 through plug board monitors to today's programmable monitors driven by a mini-computer. Central to all these monitors, however, is the concept of an 'event'. An event is the occurrence of a particular state on the system under measurement. The event we are interested in can also be a combination or a sequence of other events. An event can be as simple as an occurrence of an instruction fetch cycle or as complex as the first operand fetch cycle after executing the instruction at a certain location while executing a particular user's program. Some monitors sample a slowly varying input signal for being true or false at a certain sampling rate. The occurrence of the sampling pulse can be considered an event for the purpose of our discussion.

Since events can be so complex and since different events need to be monitored for different experiments, the event detection mechanism is the most important part of a hardware monitor. After detecting an event, the monitor can just increment a counter or update a histogram, or time stamp the event and store it in the secondary storage. In theory, it is possible to store every event with its time stamp and all the other information and later process the whole data. It is however much more economical to do some selection in the monitor itself and restrict the flow of data to only pertinent information or some condensed form of the complete information. On the other hand, if only the condensed form of information is obtained from the hardware monitor, its utility is limited to gross measurements. Event response specification is therefore another important aspect of a hardware monitoring system. Finally, the gathered data has to be presented in an understandable form for the users. Some of the data can be presented while the data collection is going on, some has to be post-processed by a computer and some data needs to be stored to build a data-base. How the data is presented is important for the monitoring system to gain acceptance.

Before we discuss the characteristics of existing hardware monitoring systems, let us briefly enumerate some of the experiments that can be performed using hardware monitors. The experiments span measurements at all system levels shown in Figure A.1. Some of the performance parameters at the upper levels are best obtained using a software monitor even though hardware monitors have been used to obtain these. We have restricted our experiments to those that need to use a hardware monitor for any of the following reasons.

1. Events at machine cycle level are not accessible to a software monitor, e.g. overlap of I/O and CPU, cache hits.

2. Software monitors are often dependent on the operating system. This makes it necessary to program separate monitors for each operating system even when the operating systems are written for the same machine. Moreover, some software monitors require the language compilers to be modified.
3. An artifact or perturbation is introduced in the measured system with any software measurement technique. This artifact cannot be ignored in some important experiments, e.g. measurements on time critical operating system functions.
4. Some measurements are prohibitively expensive if performed in software, e.g. counting instruction usage or accesses to an active data structure.
5. The hardware monitor possesses high speed counters and a high resolution timer without which most counting and timing measurements are impossible.

The applicable reasons out of these are indicated for each experiment in the figure.

Experiment	Primary reason for using hardware monitor	Event detection	Event response	Display of Information
1. Sampling events				
a> Instruction mix	2	Instruction fetch (random or each)	opcode word	Breakdown of instructions according to opcode
b> Execution profile	2	1. Instruction fetch particular user or OS 2. Overlay change	1. address from which fetched 2. New overlay number	Histogram of frequency of occurrence of instructions
c> Memory access profile	4	Memory cycle (random or each)	memory address, address space. Pc priority high or low, instruction fetch or not	Distribution of accesses in 4 address spaces X 3 pages X 2 priority levels X 2 (instruction fetch or not)
2. Counting events				
a> Memory access types	1	Read, read pause, write and write-bytes each counted by an event	increment corresponding counter	Number of reads, read-pauses, writes, and write-bytes per time interval
b> MYSN analysis	1	Memory cycle, cycles belonging to Pc, instruction fetch	increment corresponding counter	Number of cycles, cycles belonging to Pc, instruction fetches per time interval
c> Time duration of an instruction sequence	5	1. starting instruction 2.final instruction	start timer stop timer	Duration of sequence
* d> Relocation register access frequency	1 , 5	Memory cycles, O.S. instructions, access to a reloc register	increment corresponding counter	Number of relocation register accesses/kernel instructions /cycles

* applicable only to machines having relocation registers

(continued) . . .

Figure A.1 Experiments Performed Using Hardware Monitors

Experiment	Primary reason for using hardware monitor	Event detection	Event response	Display of information
2. Counting events (continued) . . .				
e> Memory contention	1 , 5	Start and end of a memory cycle	time taken to complete cycle	histogram of cycle length
f> O.S. overhead	5	1. Entry into O.S. due to	Count user instructions,	Numbers accumulated in the
g> Channel utilization and overlap	1 , 5	1. CPU busy, idle 2. hannel busy, idle	Increment one of the following counters: only CPU busy, only channel busy, both busy, both idle	Kiviat graph or numeric output
h> Cache hit/miss	1 , 5	1. Memory cycle 2. Word found in cache	Increment corresponding counter	cache hit ratio otput every second
i> Paging activity	2	1. Page read 2. Page written	increment corresponding counter	Number of pages read, written per second
3. Trace mode				
a> Device activity	2 , 1	read, write or seek command given to a device	command contents and time stamp	device activity report, including overlapped cycles
b> Time lost in a multi-processor due to contention of shared data	3 , 5	occurence of a lock, unlock or a block	lock address and time stamp	frequency of occurence of different locks, time in critical sections, lost time
c> Priority changes and blocking due to high Pc priority	1 , 5	change in Pc priority	time, address of next	duration and starting points of high priority execution

Figure A.1(continued)

Experiment	Primary reason for using hardware monitor	Event detection	Event response	Display of information
d> Detecting operating system 'hooks'	3	'write' into special 'hook' location	new 'hook' ID, time stamp	analysis of hook sequence, duration of OS functions
e> Analysis of time spent by a program running stand-alone on the machine	5 , 2	1. OS entered 2. Return to user 3. Wait started 4. Wait terminated	event ID and time stamp in each case	time spent in OS wait state and user state
f> Analysis of OS calls	4 , 2	system call enter and exit	call ID and time stamp	number of times each call is used and its time duration
g> Relocation register accesses	3 , 5	access to a relocation register	register access, type of access, time stamp	average time between read or write accesses
h> Instruction trace	4	Instruction fetch from which fetched	Opcode word and address, branch distance	Common instruction sequence, branch probality

* applicable only to machines having a relocation register

Figure A.1 (continued)

A.2.1 Event Detection

The input to a hardware monitor is the various status bits, bus lines and registers in the system under measurement (called P.host). A hardware monitor usually passively senses the values of the input signals. High impedance probes are provided so that a very small load is put on the signal under measurement by connecting the probe to it. For a general purpose monitor, the probes need to be calibrated for the specific voltage values in the P.host. There is also a problem of synchronization since the probe input is not valid when the corresponding signal is changing its state.

In the early monitors, event detection was achieved with the help of a logic plug board that consisted of an assorted collection of gates, latches and decoders. It was therefore very time consuming to set up an experiment or to switch from one experiment to another. A look at figure A.1 will show that one of the most important parts of event detection is address comparison. It is necessary to be able to detect the occurrence of a specific address or any address in a given range. It is also important to detect particular values of data being accessed or those of other probe inputs. Most modern monitors are programmable, that is, the function of the plug board is achieved by setting bits and registers in the monitor under the control of the supervisory computer (called P.sup). The hardware monitor designed and built at Carnegie-Mellon University is a programmable monitor (called K.mon) . A brief description of K.mon event detection is given below as an example of programmable monitors.

The event detecting part of K.mon is shown in Figure A.2.

The event detector senses events at the unibus cycle(i.e. memory fetch) level. It is composed of two types of modules, comparators and bit masks. A comparator has an

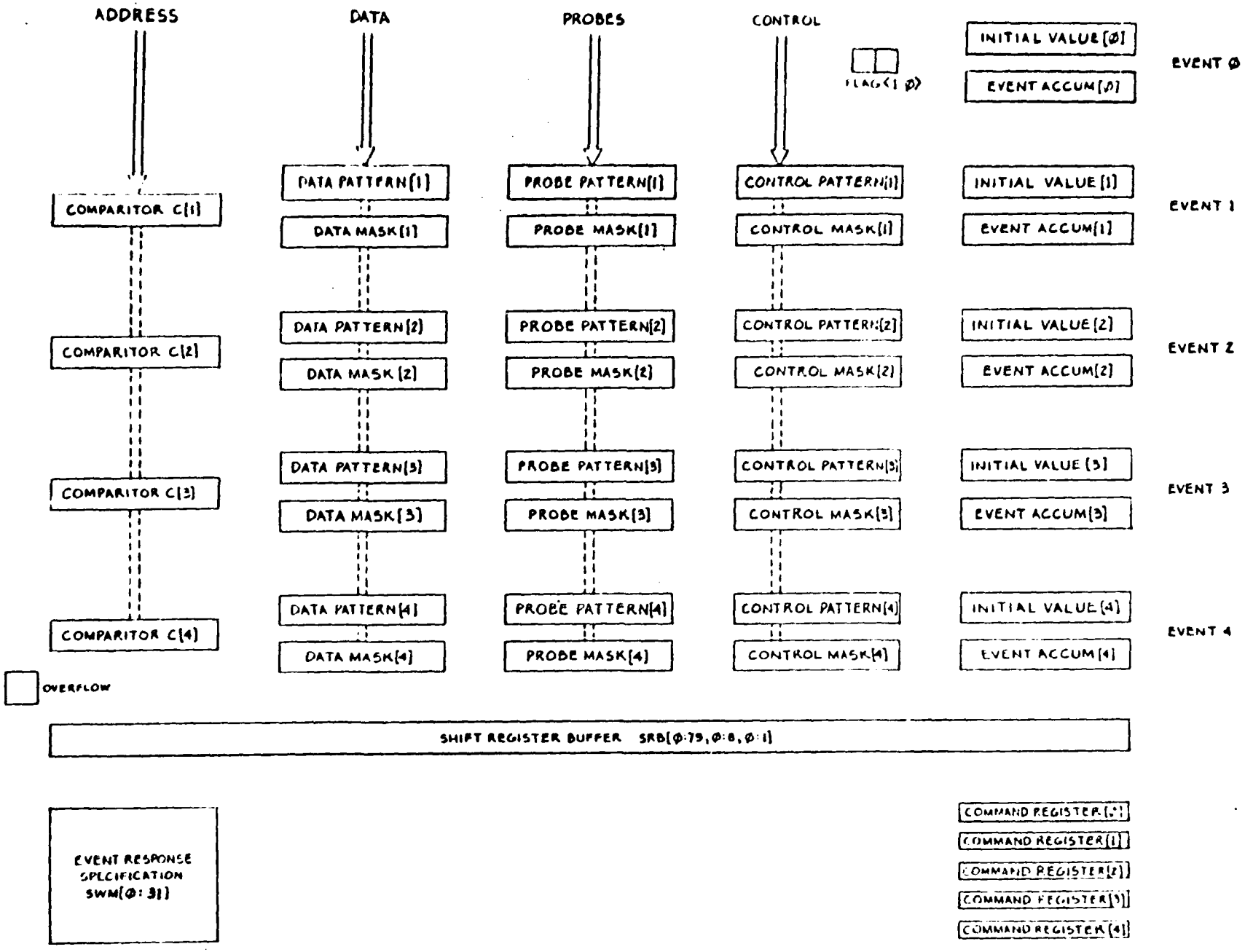


Figure A.2 Event Detection in K.mon

internal register (set by P.sup) and it produces two output signals, 'input = register' and 'input < register'. A bit mask has two internal registers (set by P.sup) . One specifies the care/ don't care conditions for each bit, the other specifies the expected bit pattern. A single output indicates if the input satisfies the specified pattern. Signal inputs for the comparators and the bit masks are arranged in four groups: unibus address, unibus data, miscellaneous probes and control. The control group includes unibus control signals. In the current implementation four events can be defined simultaneously.

In some commercial monitors, the plug boards are removable and pre-wired plug boards for certain standard experiments are provided. ADAM [HUGH74] has an interesting way of event detection. It has a monitor register which holds all the selected input signals. The monitor register is suitably masked and then compared simultaneously to sixty four bit patterns in a content addressable memory. The matching pattern determines the event that has happened.

The Waterloo hardware monitor (RCHM [MORG73]) implements a very sophisticated plug board in which some of the connections can be made under program control. For example, it contains a programmable 16 X 4 switch matrix which allows up to 4 of its 16 input signals to be available as its output. It also has programmable combinatorial logic units which accept 8 input signals and yield one output signal which is any logical function of the inputs. There is a hardware unit to detect the sequences of events. In addition to the above, it has comparators, interval timers, event/time counters and character detectors to aid event detection. Even though the hardware components are programmable, the inputs to these are usually from the plug board and the outputs are also usually available only on the plug board. This arrangement allows small changes in the experiments to be made under program control without manual intervention.

A.2.2 Event response specification

When an event is detected, it is sometimes sufficient to just record the fact whereas at other times, it is necessary to record more information like the address or data that caused the event to take place. A time stamp and the values of internal counters may also be required for later analysis. In the early monitors, only summary type information was made available. So the only response to any event was to increment an internal counter. This is sufficient if only gross average values of the measured quantities are required. If however, one needs to generate histograms for constructing analytical or simulation models, more detailed information has to be obtained by the hardware monitor. In K.mon, when an event is detected, up to 9 words of information can be obtained. These are: address, data, probes, miscellaneous signals, clock value and four words giving the number of times each of the four events has occured so far. Moreover, two internal flags can be set or reset to facilitate detecting a sequence of events. K.mon thus acts like a filter which detects certain interesting cycles out of a vast number of unibus cycles and then makes selected inputs available. This arrangement is necessary for experiments involving tracing of events. In a trace mode the input information along with the time stamp and other information internal to the monitor is transfered directly to the output storage medium.

The Waterloo monitor has a hardware time stamp register which can record 12 bits of 'environment' information plus 24 bit time when any of the 8 selected events happen. In addition it has a sequential event detector which can be programmed to recognize a sequence of events given as a regular expression.

The Tesdata MS monitor employs a 'mapping' scheme of counters which uses 4

counters to count the occurrences of the 4 possible combinations of two input signals automatically. This scheme is used to determine the overlap between CPU and a channel. The two input signals are 'CPU busy' and 'Channel busy' sampled at a certain rate. The 4 counters then count the occurrences of both CPU and channel idle, only CPU busy, only channel busy and both CPU and channel busy.

It has been recognized that quite a few of the experiments fall into the category of generating histograms, so some of the new monitors are capable of generating histograms in hardware. This reduces the amount of data going to the P.sup thereby removing the cause of a major bottleneck. The parameters of the histogram like the upper and lower bounds are generally programmable from the P.sup. Another feature that can be quite useful in certain experiments is a dynamically read/writable internal register. Such a register allows dynamic alteration of an on-going experiment in response to incoming data. As a response to an event, this register can be loaded with the current address or data values or the clock value. This register can subsequently be used in event detection and it can also be output when some other event is detected. This register can be used for tracking relocatable pieces of code and other measurements. If it is possible to store time differences in this register, then determining the maximum time difference between two events becomes easy. Hughes[HUGH74] used this technique to determine the maximum duration of the interrupt disabled mode in a system.

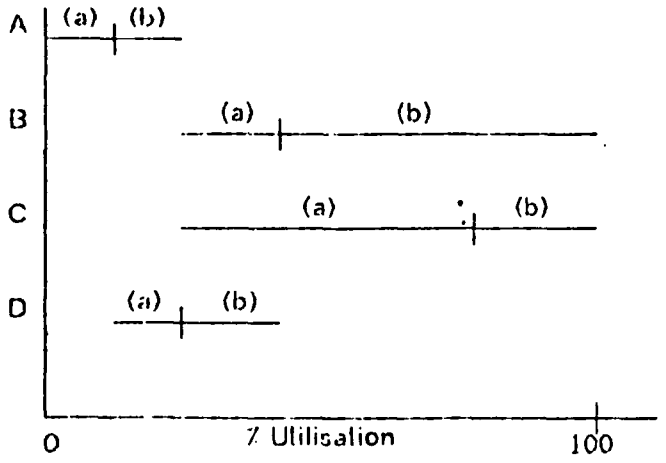
A.2.3 Display of Information

This is another area where a lot of progress has been made since the early monitors. For a summary type monitor, all that is really necessary is to display the contents of the counters, preferably in decimal. For second generation monitors,

however, it is convenient to have a tape or disk storage unit and / or a supervisory processor for on-line display. Early commercial monitors used tape storage and post-processing to generate reports. The disadvantage is that experiment set up errors are not detected until after the post-processing is done. Most modern monitors are capable of generating some real time display while also storing data for post-processing.

Some standard output formats have evolved over time. The trend has been to provide data in a pictorial format to make it easy to comprehend. Histograms are a good example of providing a visual representation of a distribution function. Gantt profiles have been used to summarize resource utilization and overlap (see figure A.3). Another representation of resource utilization is provided with a 'Kiviat graph'. It is obtained by plotting equal number of 'good' and 'bad' indicators of performance on alternate axes in a circle (see figure A.4). The Kiviat graph has a lot of visual appeal because all the utilization data is available at a glance and the shape of such a graph is an indication of the 'goodness' of a computer system.

Since quite a few experiments use a histogram to display information, some commercial monitors are equipped with special hardware to display histograms directly. On the other hand, some monitors like the Remote Controlled Hardware Monitor at Waterloo[MORG73] transfer information to a central computer over phone lines for post-processing. In some other experiments, information is gathered in a data base for long term trend analysis.



A CPU not busy
(a) System idle
(b) Channel busy and CPU idle

B CPU busy
(a) CPU busy and channel busy
(b) CPU busy and channel idle

C CPU busy
(a) Problem state
(b) Supervisor state

D Channel busy
(a) Channel busy and CPU idle
(b) CPU and channel busy

FIGURE A.3 EXAMPLE OF A GANTT PROFILE

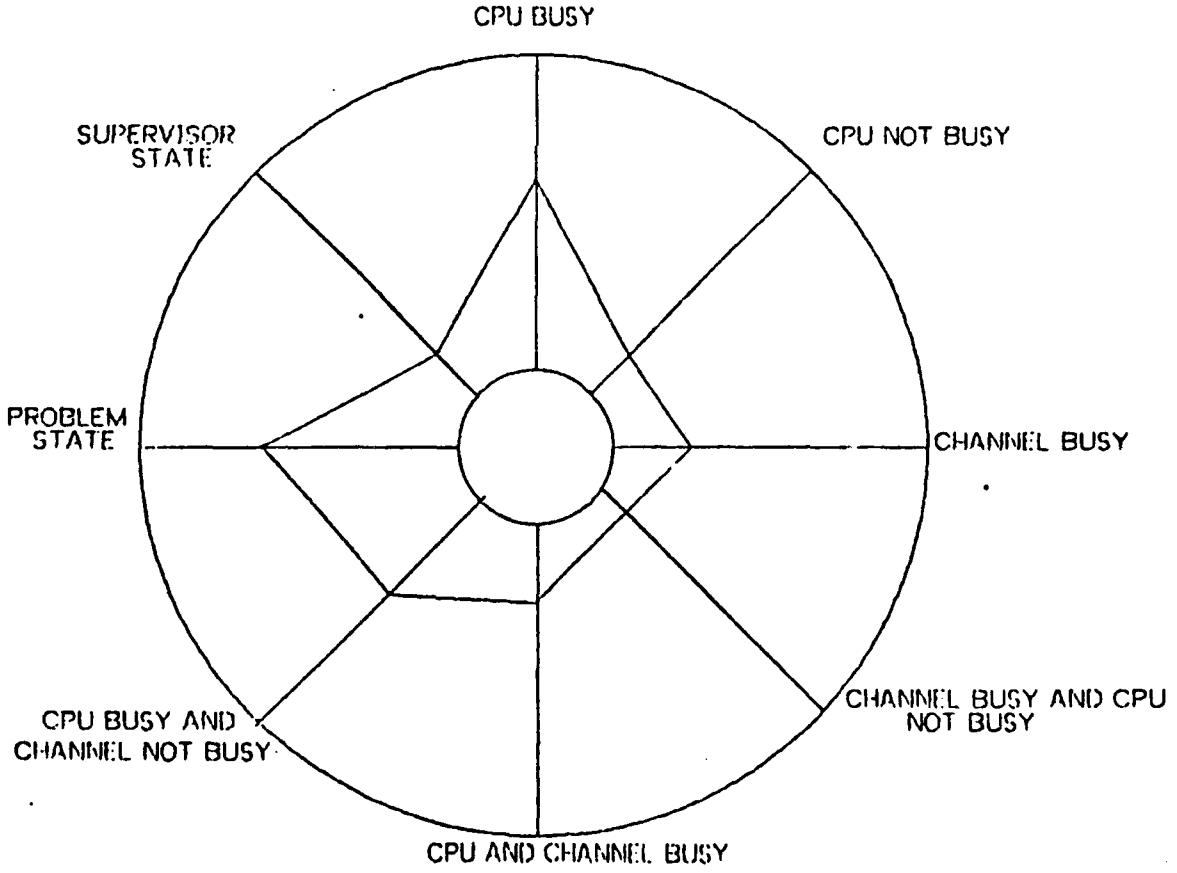


FIGURE A.4 EXAMPLE OF A KIVIAT GRAPH

A.3. Comparison of some Hardware Monitors

A few of the old and new monitors are compared below. A few commercial monitors (e.g. SUM) are no longer available and are therefore not discussed. Detailed information could not be obtained regarding the Univac 1108 monitor and a commercial monitor supplied by Computer Performance Instrumentation, Inc. The monitors discussed are listed below in approximate chronological order:

1. 7090 portable monitor [IBM63]
2. Memory bus monitor [FRYE73]
3. Neurotron Monitor [ASCH71]
4. ADAM [HUGH74]
5. K.mon [FULL73]
6. RCHM [MORG73]
7. Dynaprobe 8016 [DYNA76]
8. Tesdata MS [TESD76]

Features	7090 portable	Memory bus mon	Neuro-tron	ADAM	K.mon	RCHM	Dynaprobe 8016	Tesdata MS (model 50 and up)
Event Detection:								
1. Numbers of probes	Built in	40	36	48	64	40	20 - 160	18 - 144
2. Input repetition rate limited by probes and logic (Mhz)	0.5	1	40	2	1	5	0.1	40
3. Programmable	no	no	yes	yes	yes	partly	no	partly
4. Plug board	no	no	yes	yes	no	yes	yes	yes
5. Number of hubs	----	----	?	?	----	400	upto 1600	upto 1440
6. Number of Comparators	0	6	?	64	4	12	0	1-4
7. Bits in Comparator	----	16	?	32	16	16	------	8 - 32
8. Internal clock period for timer (microsec)	1-2	1-1000	0.025	0.1	1-4096	0.4-8	10	?
Event response specification								
9. Programmable	no	no	yes	yes	yes	yes	no	yes
10. High speed counters	5	1	32	128	4	9	16	2-8
11. Trace mode	no	yes	no	yes	yes	yes	no	yes
12. Hardware histogram	no	no	yes	yes	no	yes	yes	yes
13. Dynamically accessible register	no	no	yes	yes	no	no	no	no
Display of information								
14. Hardware Bar-graph display	no	no	yes	no	no	no	yes	yes
15. Mini processor supervisor	no	no	yes	yes	yes	yes	yes	yes
16. Output devices	Display camera	Display readout	Tape TTY	TTY, disk, CRT	TTY, telephone	TTY, telephone	Tape , numeric display	Tape, disk, CRT, Line printer graphics '
17. on-line display (hardware or supervisor program)	yes	yes	yes	yes	yes	yes	yes	yes
13. The hardware and accompanying software make it most useful to:	Hardware Architect	Systems, Applica-tions programmer	Hardware Architect	Systems, Applica-tions programmer	Hardware Architect O.S. designer	Systems, Applica-tions programmer	Systems, Applications programmer. Installation manager	Systems, Applications programmer. Installation manager

Figure A.5 A Survey of Current Monitors

A.4. Trends in the Hardware Monitor Development

The advantages of providing programmable registers in the hardware monitor over the earlier manual plug boards are obvious. With programmable registers, experiments can be modified quickly if in error or if incoming data so requires. In fact, going one step further, the hardware monitor or the host machine can be allowed to set certain internal registers (e.g. comparators or hold registers). This feature can be used to keep track of a program which can dynamically move in the main memory of the host.

Early in the development of hardware monitors it was realized that it is much more convenient to use a minicomputer to perform some of the logical and arithmetic functions rather than designing the hardware to do them. Moreover, the minicomputer can be used to store the data on secondary storage as well as communicating with the user. Depending on the rate of data gathering, the minicomputer can perform some compaction and then display or print preliminary results while the experiment is still going on.

If one is monitoring a processor on a chip, the input to the monitor is severly limited to the bus coming out of the processor. None of the internal status bits are accessible. In such cases, a self monitoring feature can be provided for microprogrammed processors. The microcode can have hooks at interesting places to enable one to insert measurement microcode. Such a monitoring system has all the hardware data in the processor available and by careful overlapping of operations it can be made to cause less perturbation than a pure software monitor. This might be a way to measure microprogrammed processors on a chip. The disadvantage is that the status bits in the peripheral devices are not directly accessible.

There is a distinct trend in providing some processing inside the hardware monitor. The examples are histogram generation or moment calculation. As more common processing requirements among different experiments are identified, it will be justifiable to put them directly in hardware. Similarly, as common display formats are discovered, they will be put in hardware or will be supplied as standard set of programs. There has been some progress in defining a measurement language. Such a language will allow the users to specify the events to be detected, the information to be gathered when an event occurs and finally the format in which the information is to be displayed.

Appendix B: The Instruction Mix Experiment

This appendix presents the output of the instruction mix experiment in detail. Since chapter 4 identifies 'MOV' as the most frequently executed instruction, we present the fractions for the execution of the 'MOV' instruction in each of the segments. Similar information can be obtained for any other instruction or addressing mode but we will not report it here for brevity. It is interesting to observe the large variation in the fractions for the MOV instruction. The fractions range from more than 0.5 in program 1 in area 5 to 0.0001 in program 4 in area 5. In fact, it can be seen that program 4 in area 5 consistently uses fewer MOV instructions than any of the other programs. We traced the reason for this behavior to the fact that it is a hand-written program which spends most of its time in a small tight containing very few MOV instructions.

In chapter 4 we reported that the variance between segments is small compared to the other two sources of variance. The data presented here clearly brings out this fact for the MOV instruction.

Fractions for the MOV instruction

APPLICATION AREA 1: Scientific Fortran Benchmarks

	Program #				
	1	2	3	4	5
Segment #					
1	.384	.317	.438	.343	.332
2	.393	.324	.469	.342	.324
3	.383	.320	.465	.329	.348
4	.392	.318	.432	.369	.332
5	.389	.313	.466	.343	.328
6	.389	.320	.440	.336	.348
7	.395	.323	.470	.345	.313
8	.387	.308	.467	.340	.325
9	.388	.322	.426	.332	.347
10	.391	.318	.469	.340	.356
11	.395	.321	.454	.358	.364
12	.385	.317	.451	.337	.361
13	.392	.322	.468	.338	.302
14	.386	.316	.439	.368	.346
15	.391	.327	.468	.341	.329
16	.389	.299	.466	.340	.335
17	.388	.317	.432	.333	.336
18	.391	.321	.464	.361	.316
19	.386	.317	.443	.346	.322
20	.390	.324	.462	.336	.349
21	.386	.323	.469	.355	.335
22	.391	.327	.422	.335	.316
23	.387	.329	.475	.336	.356
24	.393	.323	.468	.349	.335

APPLICATION AREA 2: Business Cobol Benchmarks

	Program #				
	1	2	3	4	5
Segment #					
1	.199	.306	.249	.247	.247
2	.169	.312	.256	.258	.258
3	.193	.302	.256	.223	.223
4	.152	.298	.275	.271	.271
5	.195	.302	.288	.219	.219
6	.200	.303	.247	.284	.284
7	.203	.298	.231	.248	.248
8	.250	.304	.189	.215	.215
9	.188	.303	.210	.275	.275
10	.221	.306	.239	.270	.270
11	.185	.332	.226	.196	.196
12	.188	.323	.250	.272	.272
13	.161	.248	.211	.289	.289
14	.194	.206	.229	.197	.197
15	.201	.216	.223	.249	.249
16	.194	.290	.212	.272	.272
17	.242	.265	.203	.233	.233
18	.214	.215	.225	.276	.276
19	.193	.252	.249	.288	.288
20	.211	.288	.253	.289	.289
21	.181	.274	.241	.294	.294
22	.190	.321	.249	.284	.284
23	.166	.305	.253	.289	.289
24	.193	.304	.259	.294	.294

APPLICATION AREA 3: Operating Systems

	Program # 1	2	3	4	5
Segment #					
1	.325	.313	.309	.306	.250
2	.340	.294	.324	.309	.256
3	.240	.278	.314	.307	.254
4	.295	.291	.323	.304	.258
5	.408	.329	.313	.309	.253
6	.306	.306	.313	.303	.249
7	.237	.305	.326	.313	.253
8	.381	.313	.321	.294	.256
9	.336	.302	.332	.294	.249
10	.286	.336	.315	.317	.256
11	.308	.312	.319	.311	.254
12	.323	.290	.319	.323	.251
13	.337	.313	.329	.296	.255
14	.243	.281	.317	.308	.267
15	.330	.307	.332	.307	.259
16	.406	.285	.336	.315	.253
17	.296	.311	.319	.323	.259
18	.249	.298	.306	.312	.254
19	.401	.319	.323	.305	.252
20	.335	.322	.312	.222	.252
21	.239	.306	.333	.308	.262
22	.343	.292	.323	.293	.257
23	.400	.313	.315	.294	.255
24	.311	.312	.323	.322	.236

APPLICATION AREA 4: Systems Programs

	Program # 1	2	3	4	5
Segment #					
1	.281	.339	.323	.163	.383
2	.288	.338	.320	.197	.407
3	.273	.348	.318	.208	.398
4	.277	.343	.321	.231	.407
5	.281	.350	.318	.229	.405
6	.282	.331	.241	.187	.399
7	.284	.351	.205	.165	.400
8	.285	.351	.185	.160	.403
9	.279	.347	.131	.182	.390
10	.286	.347	.135	.172	.395
11	.282	.344	.126	.202	.405
12	.289	.350	.321	.168	.412
13	.289	.352	.321	.175	.408
14	.277	.348	.320	.195	.406
15	.277	.350	.321	.210	.405
16	.275	.341	.322	.187	.412
17	.284	.341	.323	.184	.405
18	.282	.346	.313	.185	.394
19	.281	.344	.147	.194	.389
20	.279	.340	.206	.190	.381
21	.279	.348	.122	.183	.379
22	.284	.339	.138	.179	.386
23	.276	.330	.133	.187	.375
24	.277	.349	.271	.217	.406

APPLICATION AREA 5: Real Time Programs

	Program #				
	1	2	3	4	5
Segment #					
1	.470	.453	.354	.108	.403
2	.480	.450	.347	.166	.406
3	.462	.449	.361	.107	.406
4	.465	.454	.348	.140@-2	.398
5	.460	.448	.350	.450@-3	.403
6	.474	.447	.391	.250@-3	.392
7	.475	.447	.358	.000	.388
8	.474	.447	.338	.100@-3	.383
9	.465	.452	.339	.500@-3	.386
10	.460	.453	.345	.239@-1	.385
11	.472	.448	.322	.384@-1	.375
12	.477	.457	.307	.416@-1	.382
13	.459	.452	.345	.432@-1	.397
14	.490	.446	.385	.126	.402
15	.493	.448	.341	.129	.408
16	.487	.448	.363	.149	.399
17	.493	.450	.379	.129	.412
18	.483	.445	.357	.110	.410
19	.501	.449	.387	.138	.410
20	.499	.459	.356	.149	.405
21	.514	.454	.385	.140	.409
22	.512	.447	.361	.124	.401
23	.487	.453	.358	.458@-1	.395
24	.471	.453	.359	.137@-1	.368

Appendix C: Comprehensive Unibus Cycle Trace

In this appendix we present a sample output from our analysis program which analyses the Unibus cycle trace as discussed in section 6.1.4. Because of the bandwidth limitations of K.mon, it is difficult to collect a trace consisting of a large number of cycles. In this example, we were able to collect only about 20500 cycles during 30 minutes. It should be noted that the data presented here has been obtained for a single program and we cannot draw any general conclusions from this data. In fact, this particular trace shows that only about 16 percent of the instructions were MOV's whereas in chapter 4 we saw that the MOV instruction is executed on the average about 31 percent of the time.

Once a Unibus cycle trace is collected for a large number of programs, it will be possible to answer a variety of important design questions. The analysis presented here is intended to give the reader an idea of the kind of analysis that can be done with such a complete cycle trace.

Total Unibus cycles in the trace: 20520
Total cycles initiated by the processor: 20520
Breakdown according to the kind of Unibus cycle:

read	read-pause	write	write byte
17844	552	1959	165

Total instructions in the trace: 12184

The instruction mix:

name	% instr	name	% instr	name	% instr	name	% instr
mov	16.111	jsr	3.324	rts	2.594	cmp	2.511
beq	3.792	add	3.037	movb	.755	bic	.230
jmp	.213	sub	2.142	bne	2.487	tst	2.938
br	2.577	asl	6.566	dec	6.328	bpl	.140
bitb	.131	ble	.501	inc	1.182	cmpb	.435
clr	2.421	bge	.197	bit	1.543	bis	.066
nop	.000	rtt	.008	swab	1.428	bgt	5.581
rol	4.087	sxt	.000	flot	.000	bvc	.000
clrb	.542	sbcb	.000	mfpd	.000	bisb	1.888
wait	.008	neg	.295	asr	3.357	mul	.000
sob	.000	bvs	.394	comb	.000	tstb	.386
mtpd	.000	rti	.041	adc	1.773	div	.000
bcc	2.429	incb	.074	rorb	.033	unus	.000
bpt	.000	spl	.000	sbc	.886	mark	.000
ash	.000	bmi	.057	bcs	3.940	decb	.090
rolb	.287	iot	.016	com	.008	mfpi	.000
ashc	.000	bhi	1.362	emt	.000	emt	.000
emt	.000	emt	.000	trap	.000	trap	.000
trap	.000	trap	.000	negb	.000	asrb	.000
reset	.000	cond1	.025	cond2	.000	cond3	.361
cond4	.000	blt	.558	ror	7.855	mtpi	.000
xor	.000	blos	.008	adcb	.000	aslb	.000
bicb	.000						

Mode register statistics for single operand instructions

mode number	Register number							
	0	1	2	3	4	5	6	7
0	1242	1007	263	317	630	652	0	0
1	33	2	0	7	0	3	65	0
2	0	0	0	0	0	0	70	0
3	0	0	0	0	0	0	88	24
4	0	0	0	0	0	0	103	0
5	0	0	0	0	0	0	0	0
6	10	20	9	120	0	3	66	560
7	0	0	1	0	0	0	75	0

Source mode and register statistics for double operand instructions

mode number	Register number							
	0	1	2	3	4	5	6	7
0	280	175	581	294	107	94	29	0
1	2	30	27	19	57	43	6	7
2	0	0	0	0	0	3	423	404
3	0	0	0	0	0	3	0	64
4	0	0	0	0	0	0	3	0
5	0	0	0	0	0	0	0	0
6	16	52	10	0	1	5	372	383
7	0	0	0	0	0	0	21	4

Destination mode and register statistics for double operand instructions

mode number	Register number							
	0	1	2	3	4	5	6	7
0	648	515	289	170	222	116	16	1
1	9	16	69	57	42	67	132	0
2	0	0	0	0	0	0	10	201
3	0	0	0	0	0	0	0	49
4	6	0	0	0	0	0	454	0
5	0	1	0	0	0	0	0	0
6	13	2	7	1	0	1	144	233
7	0	0	0	0	1	0	6	0

Histogram of number of Unibus cycles per instruction

```
# OF SAMPLES 11854       MEAN 1.672       STD DEV. 1.038
MINIMUM  1       MAXIMUM  8               Total 19818

RANGE                    COUNT

-inf TO   1]        7591     ********************
(  1 TO   2]        1735     ****
(  2 TO   3]        1530     ****
(  3 TO   4]        848      **
(  4 TO   5]        131
(  5 TO   6]         14
(  6 TO   7]          4
(  7 TO   8]          1
(  8 TO   9]          0
(  9 TO  10]          0
( 10 TO +inf                  0
---------------------------------------
```

Histogram of the index values off stack pointer:

This information can be used to decide the ulility of providing a small field in every instruction to specify the parameter number off the stack pointer.

```
# OF SAMPLES 578MEAN  23.159      STD DEV.  172.455
MINIMUM   0       MAXIMUM 3072              Total  13386

RANGE                       COUNT

-inf TO -16]          0
(-16 TO -15]          0
(-15 TO -14]          0
(-14 TO -13]          0
(-13 TO -12]          0
(-12 TO -11]          0
(-11 TO -10]          0
(-10 TO  -9]          0
( -9 TO  -8]          0
( -8 TO  -7]          0
( -7 TO  -6]          0
( -6 TO  -5]          0
( -5 TO  -4]          0
( -4 TO  -3]          0
( -3 TO  -2]          0
( -2 TO  -1]          0
( -1 TO   0]          1
(  0 TO   1]         61       *********
(  1 TO   2]         74       **********
(  2 TO   3]          0
(  3 TO   4]         32       *****
(  4 TO   5]          0
(  5 TO   6]         48       *******
(  6 TO   7]          0
(  7 TO   8]         47       *******
(  8 TO   9]          0
(  9 TO  10]         91       **************
( 10 TO  11]          0
( 11 TO  12]         34       *****
( 12 TO  13]          0
( 13 TO  14]         39       ******
( 14 TO  15]          0
( 15 TO  16]         28       ****
( 16 TO +inf        123       ********************
-----------------------------------------

-------------------------------------------------------------------------------
```

Histogram of the immediate mode operands:

This information can be used to decide the utility of providing a small field in every instruction to specify small immediate operands.

OF SAMPLES 602MEAN 801.585 STD DEV. 5525.045
MINIMUM -32764 MAXIMUM 32768 Total 482554

RANGE	COUNT	
-inf TO -16]	45	**
(-16 TO -15]	0	
(-15 TO -14]	0	
(-14 TO -13]	0	
(-13 TO -12]	0	
(-12 TO -11]	0	
(-11 TO -10]	0	
(-10 TO -9]	0	
(-9 TO -8]	10	
(-8 TO -7]	1	
(-7 TO -6]	0	
(-6 TO -5]	1	
(-5 TO -4]	3	
(-4 TO -3]	1	
(-3 TO -2]	3	
(-2 TO -1]	0	
(-1 TO 0]	0	
(0 TO 1]	57	***
(1 TO 2]	25	*
(2 TO 3]	2	
(3 TO 4]	16	
(4 TO 5]	1	
(5 TO 6]	1	
(6 TO 7]	17	
(7 TO 8]	9	
(8 TO 9]	9	
(9 TO 10]	7	
(10 TO 11]	0	
(11 TO 12]	3	
(12 TO 13]	0	
(13 TO 14]	4	
(14 TO 15]	27	*
(15 TO 16]	16	
(16 TO +inf	344	**********

Histogram of the index values off registers other than SP

OF SAMPLES 1437 MEAN -3890.991 STD DEV. 10350.534
MINIMUM -32694 MAXIMUM 32722 Total -5591354

RANGE	COUNT	
-inf TO -16]	950	**********
(-16 TO -15]	0	
(-15 TO -14]	0	
(-14 TO -13]	0	
(-13 TO -12]	0	
(-12 TO -11]	0	
(-11 TO -10]	0	
(-10 TO -9]	0	
(-9 TO -8]	0	
(-8 TO -7]	0	
(-7 TO -6]	0	
(-6 TO -5]	0	
(-5 TO -4]	10	
(-4 TO -3]	0	
(-3 TO -2]	2	
(-2 TO -1]	0	
(-1 TO 0]	0	
(0 TO 1]	0	
(1 TO 2]	18	
(2 TO 3]	0	
(3 TO 4]	7	
(4 TO 5]	14	
(5 TO 6]	13	
(6 TO 7]	0	
(7 TO 8]	5	
(8 TO 9]	0	
(9 TO 10]	8	
(10 TO 11]	0	
(11 TO 12]	9	
(12 TO 13]	2	
(13 TO 14]	10	
(14 TO 15]	0	
(15 TO 16]	1	
(16 TO +inf	388	********

Appendix D: The Execution Profile for Hydra

The following is the execution profile of Hydra measured while executing a parallel root finding program utilizing kernel semaphores for synchronization. The Hydra kernel instructions were sampled at random using the hardware monitor. The changes in the overlay code page were also monitored with the hardware monitor.

Relocation register	function	Number of instructions
0	stack page	0
1	common data	82
2	overlay data	0
3	overlay data	0
4	overlay code	10840
5	common code	3987
6	local memory	2739
7	device registers	0

Total number of instructions sampled: 17648

The above table shows some instructions being executed out of a data page via relocation register 1. This is not due to any error in the hardware monitor but Hydra implementors found it convenient to execute some instructions from a page which otherwise contains only common shared data. Hydra consists of about 50 pages of instructions and data. By monitoring the changes in the relocation register 4, we were able to identify which of the overlay code pages (from page 7 through page 47 in the following listing) was being accessed through that register. In the following listing, routines that do not have any samples in them are suppressed. For the common code page (page 6),however, all global routine names are printed.

Page number 6 Total Instructions: 3987

address	routine name	number of samples
120000	CSUS.C	0
120006	ERRPRO	0
120232	HLNK.C	0

120252	$LINK	363
120274	$EHY11	0
120354	$EHY01	0
120460	$EHYXT	0
120502	TTTC.C	0
120532	TTWRIT	0
120574	OUTCH	0
120632	SIXOUT	0
120674	MOVE	7
120732	MOVE16	1
120734	MOVE15	0
120736	MOVE14	0
120740	MOVE13	0
120742	MOVE12	0
120744	MOVE11	0
120746	MOVE10	1
120750	MOVE9	1
120754	MOVE7	1
120756	MOVE6	0
120760	MOVE5	1
120764	MOVE3	1
120766	MOVE2	1
120770	MOVE1	0
120772	MOVE0	7
121006	MOVE4	0
121030	MOVE8	27
121062	PGCM.C	0
121074	STCM.C	0
121076	GETPAG	0
121124	RETPAG	0
121162	SETHDR	0
121266	GTSZ1	0
121330	GTSZ2	0
121374	GTYP1	3
121430	GTYP2	0
121460	IDLE.C	0
121462	IDLEDI	0
121502	IDLE	0
121740	IDLE.P	0
121770	LVEC.C	0
122010	LVEC.P	0
122046	FPCM.C	0
122156	OBJSHR	60
122250	OBJDEL	65
122410	COBJDE	1
122502	OBJADE	0
122562	OBJASH	0
122646	FPSEM	2
122706	FVSEM	3
122766	FCSEM	0

123026	FCMUT	42
123066	FPMUT	5
123126	FVMUT	72
123240	WHTOBJ	0
123304	FPCHK	0
123322	WHTTOT	0
123334	WHTACT	0
123346	FPOB	0
123350	TYPEMA	2
123534	SHRTYP	0
123564	OBJTYP	1
123626	FTYPE	0
123660	TYPINC	4
123722	TYPDEC	0
124014	ATYPDE	0
124106	WHTYPE	0
124156	TYHS	0
124160	DPART2	253
124270	DPART1	101
124412	RDPRT2	0
124526	RDPRT1	2
124564	DATA2M	33
125020	DATA1M	3
125064	RDATA2	1
125242	RDATA1	0
125306	XITEM2	17
125550	XITEM1	4
125614	ITEM2M	582
126000	ITEM1M	7
126044	WITEM2	0
126224	WITEM1	0
126270	DATALE	0
126370	ITEMLE	10
126556	MSCM.C	0
126560	FIRSTO	1
126630	DOCHKS	0
126712	BADMEM	0
126724	ITCM.C	20
127032	TYPOBJ	0
127072	PROCOB	1
127132	LNSOBJ	0
127172	POLICY	2
127232	PRCSOB	1
127272	PGOBJ	0
127332	SEMOBJ	80
127372	PSEMOB	0
127432	DATAOB	0
127472	PORTOB	1
127532	DEVOBJ	0
127572	UNIVOB	0

127632	EQTYPE	0
127730	EQREF	0
130042	FITYPE	0
130110	SITYPE	0
130122	FPRT1	362
130240	FPRT2	1
130352	ANDRTS	0
130502	ORRTS	0
130562	MOVRTS	1
130666	RTSBST	0
130754	CHRTSB	0
131046	CHKRTS	1
131136	PASSIT	0
131172	MAKNUL	3
131212	SHARIT	25
131470	DLTITM	14
131650	SETMPL	0
131736	MKITM	0
132054	MAKITM	0
132136	OBJC	0
132216	ITCM.P	0
132240	CDIT.C	0
132414	CALDEB	0
132442	CKMP.C	1
132462	TSTLOC	0
132502	LOCK	315
132612	IPI7	178
133004	BDLOK	0
133020	BDULOK	0
133034	UNLOCK	388
133106	PCBCUR	2
133136	PRCSID	218
133342	INCRIT	59
133356	DNCRIT	146
133416	PMUTEX	6
133460	CONDPM	33
133552	VMUTEX	69
133642	P	39
133670	CONDP	32
133724	V	66
133772	CKMP.P	0
134106	DMNC.C	0
134110	TGSTDM	0
134130	SIGE.C	0
134134	$SIGN1	1
134154	$ENABL	184
134216	KL61.C	0
134234	SIX005	0
134354	KM61.C	0
134710	KM61.P	0

134734	KCHK.C	0
134736	CHKADR	43
135034	ASA7.C	0
135224	UADR,C	4
135250	CHKUAD	2
135416	CHKUBY	0
135546	USER1M	2
135600	USER2M	0
135632	SETUDI	0
135650	UADR	2
136324	GETGNA	0
136414	READGC	0

Page number 7 Total Instructions: 52

100000	FPOB.C	5
100406	GETAFP	1
101066	OBJCRE	22
102404	OBJDAT	6
104174	WHATDA	1
104362	STORDA	1
105006	COPYDA	1
110456	ACTPP	2
112220	PASOBJ	13

Page number 10 Total Instructions: 22

100002	GTPFPA	2
100720	GETPFP	7
104344	GETPPR	2
105016	FREEFP	3
106566	FREEGS	2
115024	TRUENU	6

Page number 11 Total Instructions: 23

107754	KMIT.C	5
110524	STORPC	1
111312	POLRCV	4
112106	SETPOL	1
112772	PORVLN	10
113742	STOPCU	1
115202	PRCSIC	1

Page number 12 Total Instructions: 392

101126	CHKLST	305
102420	NGETCO	72
103676	NRETCO	15

Page number	13	Total Instructions:	5
110264		HASH	2
110452		HTLINK	1
115414		INCFPA	1
116106		SUBDPS	1
Page number	22	Total Instructions:	309
100000		IOTN.C	293
103366		TRCKMP	16
Page number	23	Total Instructions:	781
100000		MKRN.C	722
104504		POLCY	1
105502		RCVPOL	2
107270		RSTREG	2
112776		SKRN.C	54
Page number	24	Total Instructions:	841
100002		NXTLOC	2
101406		COMPAR	1
102234		GETDAT	1
102524		PUTDAT	1
105452		STORE	2
106062		LOAD	9
107530		DELETE	7
111220		CWLK.C	199
112116		V4WALK	291
117020		PSEM	215
117400		VSEM	113
Page number	25	Total Instructions:	35
103452		START	2
105144		PSHCON	1
107356		LNS.C	1
107636		MERGE	2
111574		SETUPL	22
113162		DOCALL	1
114246		KRETUR	3
115236		TCALL	3
Page number	26	Total Instructions:	93
100000		IOTK.C	3
100042		MAPHT	2
100072		UNMAPH	6
100742		GETB	1

104036	ENQBEF	3
104172	DEQ	14
104320	QAPPEN	7
104442	CQAPPE	2
104736	QREMOV	14
107104	IOSEND	1
110126	DOIO	9
110426	RGETOP	2
110634	RGETBU	17
111650	RGETIN	4
116304	PRPRF1	8

Page number 30 Total Instructions: 732

100000	IORP.C	337
104220	PRPRP1	2
104710	UPRP11	95
105336	IORP.P	298

Page number 31 Total Instructions: 343

100000	FEND.O	1
100034	FEERCT	66
102174	FEND.C	54
103426	FEOIR	86
104320	FEDISC	6
104546	FEDOWN	2
105122	FEUP	10
105506	ASAI.C	1
106110	PRASA	1
107630	UPASA	1
107766	ASAIO	13
112276	INTWIM	1
112416	STIMP	70
113342	PRPIMP	2
113700	UPIMP	2
114272	GENIPI	26
114700	FASTSE	1

Page number 34 Total Instructions: 114

100152	MSRV.C	46
104320	PACC.C	20
112076	MCREAT	1
112374	MREA.C	20
112674	MREAD	1
113172	MWRI.C	16
113472	MWRITE	1
113772	MSND.C	8
115400	MRSVP	1

Page number 35	Total Instructions:	13
102366	MRPY.C	1
103140	MREPLY	1
104276	MRCV.C	7
105730	MWAIT	3
112310	VPOLSE	1

Page number 44	Total Instructions:	7084
100000	KMPS.C	149
100034	DELINK	257
100144	ENQ	1021
100542	ENDPRC	144
100726	REQPRO	981
101152	HIGHPJ	337
101306	GETSPA	1
101376	FREESP	2
101446	PRIWIN	74
101470	ADDTIM	76
101524	SUBTIM	81
102234	INIWAT	700
102642	SWCXT	249
103174	SWAPTO	340
103550	SELCTE	1009
103662	SELCTS	275
104024	IPSCHE	271
105244	RETHIN	121
105414	INITSE	623
106564	SEND	1
106722	RECEIV	3
107254	CLOCK	61
110150	KMPA.C	1
110504	SENDST	3
111460	STARTP	20
112714	KSTOPC	1
113626	RESCHE	1
114136	TIMSCH	1
115206	DELPRE	6
116270	PRSTRT	2
116362	PRDELP	2
116454	PRSEM	120
116714	PRPX	58
116762	PRVX	38
117034	PRCPX	55

Page number 45	Total Instructions:	6
140000	TRPS.O	6

Page number 46	Total Instructions: 267	
145000	LSUS.C	154
145222	MCLOCK	5
145264	IPCLOC	7
145300	MSCHED	14
145352	IPI4	32
145444	MIOT	55

Page number 47	Total Instructions: 2466	
153000	I.MID.C	116
153176	DRTI.C	5
154160	MULD.C	18
154550	SAVR.C	548
154564	$SAV3	273
154602	$SAV4	191
154622	$SAV5	1156
154770	SIX12	78
155462	XSIX12	81

Routine names in the order of decreasing samples:

This list is not normalized according to size of the routines so larger routines may contain more samples even though they are not executed very often. Only routines with more than 10 instruction samples are listed. Information presented here can be used to decide which routines should be moved to the common code page (accesses via RR5) and which can be put in the overlay pages without excessive cost.

Routines in the common code page are marked with *
Routines in the local memory are marked with L

Page num	Address	Routine	(size)	Number of samples
47	154622 L	$SAV5	(28)	1156
44	100144	ENQ	(254)	1021
44	103550	SELCTE	(74)	1009
44	100726	REQPRO	(148)	981
23	100000	MKRN.C	(2132)	722
44	102234	INIWAT	(262)	700
44	105414	INITSE	(450)	623
6	125614 *	ITEM2M	(116)	582
47	154550 L	SAVR.C	(12)	548
6	133034 *	UNLOCK	(42)	388
6	120252 *	$LINK	(18)	363

6	130122 *	FPRT1	(78)	362
44	103174	SWAPTO	(170)	340
44	101152	HIGHF1	(46)	337
30	100000	IORP.C	(2006)	337
6	132502 *	LOCK	(72)	315
12	101126	CHKLST	(698)	305
30	105336	IORP.P	(5409)	298
22	100000	IOTN.C	(482)	293
24	112116	V4WALK	(120)	291
44	103662	SELCTS	(98)	275
47	154564 L	$SAV3	(14)	273
44	104024	IPSCHE	(572)	271
44	100034	DELINK	(30)	257
6	124160 *	DPART2	(72)	253
44	102642	SWCXT	(218)	249
6	133136 *	PRCSID	(132)	218
24	117020	PSEM	(154)	215
24	111220	CWLK.C	(446)	199
47	154602 L	$SAV4	(16)	191
6	134154 *	SENABL	(34)	184
6	132612 *	IPI7	(122)	178
46	145000 L	LSUS.C	(146)	154
44	100000	KMPS.C	(28)	149
6	133356 *	DNCRIT	(32)	146
44	100542	FNDPRC	(116)	144
44	105244	RETHIN	(104)	121
44	116454	PRSEM	(160)	120
47	153000 L	LMID.C	(126)	116
24	117400	VSEM	(88)	113
6	124270 *	DPART1	(82)	101
30	104710	UPRP11	(278)	95
31	103426	FEOJR	(442)	86
47	155462 L	XSIX12	(86)	81
44	101524	SUBTIM	(328)	81
6	127332 *	SEMOBJ	(32)	80
47	154770 L	SIX12	(264)	78
44	101470	ADDTIM	(28)	76
44	101446	PRIWIN	(18)	74
12	102420	NGETCO	(686)	72
6	123126 *	FVMUT	(74)	72
31	112416	STIMP	(468)	70
6	133552 *	VMUTEX	(56)	69
31	100034	FEERCT	(1120)	66
6	133724 *	V	(38)	66
6	122250 *	OBJDEL	(96)	65
44	107254	CLOCK	(332)	61
6	122156 *	OBJSHR	(58)	60
6	133342 *	INCRIT	(12)	59
44	116714	PRPX	(38)	58
46	145444 L	MIOT	(60)	55

44	117034	PRCPX	(42)	55
31	102174	FEND.C	(666)	54
23	112776	SKRN.C	(78)	54
34	100152	MSRV.C	(1994)	46
6	134736 *	CHKADR	(62)	43
6	123026 *	FCMUT	(32)	42
6	133642 *	P	(22)	39
44	116762	PRVX	(42)	38
6	133460 *	CONDPM	(58)	33
6	124564 *	DATA2M	(156)	33
46	145352 L	IPI4	(58)	32
6	133670 *	CONDP	(28)	32
6	121030 *	MOVE8	(26)	27
31	114272	GENIPI	(208)	26
6	131212 *	SHARIT	(174)	25
25	111574	SETUPL	(758)	22
7	101066	OBJCRE	(614)	22
44	111460	STARTP	(668)	20
34	112374	MREA.C	(192)	20
34	104320	PACC.C	(266)	20
6	126724 *	ITCM.C	(70)	20
47	154160 L	MULD.C	(164)	18
26	110634	RGETBU	(524)	17
6	125306 *	XITEM2	(162)	17
34	113172	MWRI.C	(192)	16
22	103366	TRCKMP	(230)	16
12	103676	NRETCO	(716)	15
46	145300 L	MSCHED	(42)	14
26	104736	QREMOV	(82)	14
26	104172	DEQ	(86)	14
6	131470 *	DLTITM	(112)	14
31	107766	ASAIO	(1224)	13
7	112220	PASOBJ	(1516)	13
31	105122	FEUP	(244)	10
11	112772	PORVLN	(182)	10
6	126370 *	ITEMLE	(118)	10

Appendix E: RSX11-M Major Processing Functions Trace

In this appendix we present a short sample from the trace obtained for our study of RSX11-M. It is intended to give the reader an idea of how a hardware monitor can be used to gather a comprehensive trace of the activities happening inside an operating system. It should be noted that this trace was obtained while executing a simple command typed at a terminal. It is very interesting to find that such a lot of activity goes on in the operating system even to process a simple request from a terminal.

Event trace for RSX11M:

Time since the beginning (microsec)	Processor cycles since the beginning	Description of the event
4088913	121926	Begin TTY input interrupt
4089247	122148	End interrupt service
4089538	122181	Begin TTY output interrupt
4089765	122332	End interrupt service
4126177	123135	Begin TTY output interrupt
4126341	123246	Begin $fork
4126371	123269	End interrupt service
4126457	123327	Begin TTY interrupt fork
4127068	123745	Begin TTY driver output
4127107	123769	End TTY output initiation
4127153	123800	Honor reschedule request
4127192	123825	Begin context switch
4127272	123881	>>>>>> Context switch to: LOADER
4127534	124058	End context switch
4127892	124303	Begin EMT processing
4128133	124464	$$$$$$ EMT code: 1 QUEUE I/O
4128208	124513	Begin QIO processing
4128940	124989	Begin RP04 driver initiation
4129792	125551	****** Read on unit 0, word count = 2048, cylinder movement of 3
4129797	125555	End RP04 driver processing
4129919	125635	Begin EMT processing
4130149	125789	$$$$$$ EMT code: 41 Wait for single event flag
4130371	125925	Honor reschedule request
4130623	126021	Begin context switch
4130768	126077	>>>>>> Context switch to: LV4
4131144	126254	End context switch
4136291	126319	Begin RP04 interrupt processing

4136326	126345	Begin $fork
4136357	126368	End interrupt service
4136443	126426	Begin RP04 fork processing
4136829	126679	RP04 request task reschedule
4136837	126685	Begin RP04 driver initiation
4136880	126709	End RP04 driver processing
4136922	126740	Honor reschedule request
4136990	126786	Begin context switch
4137070	126842	>>>>>> Context switch to: LOADER
4137332	127019	End context switch
4139096	128193	Honor reschedule request
4139135	128218	Begin context switch
4139157	128233	>>>>>> Context switch to: Monitor Control Routine
4139420	128410	End context switch
4140080	128846	Begin EMT processing
4140309	129000	$$$$$$ EMT code: 1 QUEUE I/O
4140384	129049	Begin QIO processing
4141197	129578	Begin RP04 driver initiation
4142166	130220	****** Read on unit 0, word count = 932, cylinder movement of 0
4142172	130224	End RP04 driver processing
4142426	130393	Begin EMT processing
4142656	130547	$$$$$$ EMT code: 41 Wait for single event flag
4142865	130686	Honor reschedule request
4143010	130782	Begin context switch
4143090	130838	>>>>>> Context switch to: LV4
4143360	131015	End context switch

Appendix F: The Device Utilization Experiment

This appendix presents the output of our device activity analysis program. The hardware monitor was used to monitor all the 'write's into any of the device registers on the PDP-11 Unibus. Even though only two units of a device were active during the data collection, the program is capable of analysing the activities of all the units of all the devices present on the host system. Since PDP-11 architecture does not employ channels for I/O, there is no direct analogue for the quantity 'CPU and Channel Busy' which is a very popular measure of component overlap in other systems. We have defined another measure for PDP-11's which can be used to determine the overlap between I/O activity and processor activity. We count the number of cycles initiated by the processor from the time a disk is given an I/O command ('GO' bit is set in the controller) until the completion interrupt is received from the device.

It should be noted that due to limitations in the output bandwidth of K.mon, we could not get a continuous trace of device activity. Some small discrepancies are therefore present in the data reported here e.g.for unit 1 of the disk the number of write check transfers is larger than the number of write transfers. This probably arises because a write operation was missed while the hardware monitor was recovering from an overflow but the following write check operation was successfully traced.

Total time of measurement : 46630.576 millisec
Total cycles on the unibus: 4890778
Total processor cycles : 4351877
Total device cycles in actual transfers: 427176
Overlapped cycles between CPU and RP11: 431103

DISK ACTIVITY DISTRIBUTION FOR RP11
Total number of transfers = 140

	Unit number 0	1	2
Disk Reads.....	54272	90112	0
Disk Writes....	480	142568	0
Disk Write checks....	480	143360	0

Disk arm movement histogram

```
  OF SAMPLES  86      MEAN  43.744     STD DEV.  61.442
MINIMUM   1   MAXIMUM 381              Total  4278

RANGE                    COUNT

-inf TO   0]               0
(  0 TO   5]              26      **************************
(  5 TO  10]               0
( 10 TO  15]               2      *
( 15 TO  20]               4      ****
( 20 TO  25]               4      ****
( 25 TO  30]               6      ******
( 30 TO  35]               1
( 35 TO  40]               5      ****
( 40 TO  45]               5      ****
( 45 TO  50]               0
( 50 TO  55]               3      ***
( 55 TO  60]               2      *
( 60 TO  65]               2      *
( 65 TO  70]               0
( 70 TO  75]               2      *
( 75 TO  80]               1
( 80 TO  85]               0
( 85 TO  90]               1
( 90 TO  95]               0
( 95 TO 100]              16      ****************
(100 TO 105]               0
(105 TO 110]               0
(110 TO 115]               1
(115 TO 120]               0
(120 TO 125]               0
(125 TO 130]               0
(130 TO 135]               1
(135 TO 140]               0
(140 TO 145]               0
(145 TO 150]               0
(150 TO 155]               0
(155 TO 160]               0
(160 TO 165]               0
(165 TO 170]               0
(170 TO 175]               0
(175 TO 180]               0
(180 TO 185]               0
(185 TO 190]               1
(190 TO 195]               0
(195 TO 200]               0
```

(200 TO 205]	0
(205 TO 210]	0
(210 TO 215]	0
(215 TO 220]	0
(220 TO 225]	1
(225 TO 230]	0
(230 TO 235]	0
(235 TO 240]	0
(240 TO 245]	1
(245 TO 250]	0
(250 TO 255]	0
(255 TO 260]	0
(260 TO 265]	0
(265 TO 270]	0
(270 TO 275]	0
(275 TO 280]	0
(280 TO 285]	0
(285 TO 290]	0
(290 TO 295]	0
(295 TO 300]	0
(300 TO 305]	0
(305 TO 310]	0
(310 TO 315]	0
(315 TO 320]	0
(320 TO 325]	0
(325 TO 330]	0
(330 TO 335]	0
(335 TO 340]	0
(340 TO 345]	0
(345 TO 350]	0
(350 TO 355]	0
(355 TO 360]	0
(360 TO 365]	0
(365 TO 370]	0
(370 TO 375]	0
(375 TO 380]	0
(380 TO 385]	1
(385 TO 390]	0
(390 TO 395]	0
(395 TO 400]	0
(400 TO +inf	0

Disk transfer time histogram (milli sec)

```
  OF SAMPLES 140      MEAN  43.300     STD DEV.  18.103
MINIMUM   2  MAXIMUM  65               Total  6061.936

RANGE                  COUNT

-inf TO   0]             0
(  0 TO   2]             0
(  2 TO   4]             3      ***
(  4 TO   6]             2      *
(  6 TO   8]             4      ***
(  8 TO  10]             2      *
( 10 TO  12]             2      *
( 12 TO  14]             1
( 14 TO  16]             3      ***
( 16 TO  18]             5      *****
( 18 TO  20]             2      *
( 20 TO  22]             0
( 22 TO  24]             2      *
( 24 TO  26]             9      *********
( 26 TO  28]             3      ***
( 28 TO  30]             1
( 30 TO  32]             0
( 32 TO  34]             0
( 34 TO  36]             0
( 36 TO  38]             0
( 38 TO  40]             0
( 40 TO  42]             4      ***
( 42 TO  44]             5      *****
( 44 TO  46]             6      ******
( 46 TO  48]             8      ********
( 48 TO  50]            12      ************
( 50 TO  52]             7      *******
( 52 TO  54]             6      ******
( 54 TO  56]             6      ******
( 56 TO  58]            28      ****************************
( 58 TO  60]             2      *
( 60 TO  62]             5      *****
( 62 TO  64]             6      ******
( 64 TO  66]             6      ******
( 66 TO  68]             0
( 68 TO  70]             0
( 70 TO  72]             0
( 72 TO  74]             0
( 74 TO  76]             0
( 76 TO  78]             0
( 78 TO  80]             0
```

(80 TO 82]	0
(82 TO 84]	0
(84 TO 86]	0
(86 TO 88]	0
(88 TO 90]	0
(90 TO 92]	0
(92 TO 94]	0
(94 TO 96]	0
(96 TO 98]	0
(98 TO 100]	0
(100 TO +inf	0

Disk word count histogram

```
  OF SAMPLES 140      MEAN  3051.257  STD DEV.  1688.252
MINIMUM 256  MAXIMUM 4096              Total  427176

RANGE                  COUNT

-inf TO   0]             0
(  0 TO 256]             0
(256 TO 512]            22      ******
(512 TO 768]            17      *****
(768 TO 1024]            0
(1024 TO 1280]           0
(1280 TO 1536]           0
(1536 TO 1792]           0
(1792 TO 2048]           0
(2048 TO 2304]           0
(2304 TO 2560]           0
(2560 TO 2816]           0
(2816 TO 3072]           0
(3072 TO 3328]           0
(3328 TO 3584]           0
(3584 TO 3840]           0
(3840 TO 4096]           0
(4096 TO +inf          101      ****************************
----------------------------------------

------------------------------------------------------------
```